Настройка операторов SQL в Microsoft SQL Server 2000

Kevin Kline, Claudia Fernandez

Quest Software, Inc

Введение

Эта статья охватывает базовые методы, используемые для настройки операторов SELECT в Microsoft SQL Server – реляционной СУБД. Мы обсудим возможности, предоставляемые графическим интерфейсом в Microsoft SQL Enterprise Manager или в Microsoft SQL Query Analyzer, а также дадим краткий обзор средствам настройки запросов Quest Software.

В добавление к методам настройки, мы опишем несколько лучших практических приемов, которые вы сможете использовать в операторах SELECT, для улучшения их производительности. Все примеры и синтаксис протестированы на Microsoft SQL Server 2000. Эта статья даст вам основные представления об инструментах настройки запросов и технических приемах, предоставляемых инструментарием Microsoft. Мы охватим ряд технических приемов построения запросов, повышающих производительность и скорость считывания данных.

SQL Server посредством некоторых команд SET и DBCC позволяет оценить производительность транзакций с помощью выборки активности ввода-вывода и времени выполнения. Кроме того, некоторые команды DBCC могут использоваться для получения более детальных разъяснений по некоторым статистикам индекса, оценке стоимости каждого возможного плана выполнения и повышения производительности. Более детально команды SET и DBCC описаны в официальном издании “Analyzing and Optimizing T-SQL Query Performance on Microsoft SQL Server using SET and DBCC” («Анализ и оптимизация производительности запросов T-SQL в Microsoft SQL Server с использованием SET и DBCC»), первом из серии четырех официальных изданий по настройке производительности.

Методы настройки Microsoft

Microsoft предоставляет три основных способа настройки запросов:

1. контроль операций чтения и записи при выполнении запроса с использованием *SET STATISTICS IO*;

2. контроль времени выполнения запроса с использованием *SET STATISTICS TIME*;

3. анализ плана выполнения запросов с использованием *SET SHOWPLAN*.

Этот документ в начале покажет, как использовать эти три технологии от Microsoft, затем, будут продемонстрированы экспертные технологии, доступные с помощью инструментов фирмы Quest Software.

SET STATISTICS IO

Команда *SET STATISTICS IO ON* предписывает SQL Server предоставить отчет о реальной активности ввода-вывода при выполнении транзакций. Эта команда не может использоваться совместно с опцией *SET NOEXEC ON*, т.к. она имеет смысл только для контроля активности команд ввода-вывода, которые фактически выполняются. Как только опция включена, каждый запрос генерирует дополнительные выходные данные, которые содержат статистику ввода-вывода. Для отключения этой опции используется команда *SET STATISTICS IO OFF*.

Эти команды также работают в Sybase Adaptive Server, хотя некоторые результирующие множества могут выглядеть несколько иначе.

Например, следующий сценарий получает статистические данные об операциях ввода-вывода для простого запроса, подсчитывающего количество строк в таблице *employees* в базе данных *northwind*:

```
SET STATISTIC IO ON
GO
SELECT COUNT(*) FROM employees
GO
SET STATISTICS IO OFF
GO
```

Результаты:

```
-----------
2977
Table 'Employees'. Scan count 1, logical reads 53, physical reads 0, read-
ahead reads 0.
```

Здесь Scan count показывает число выполненных сканирований. Logical reads – число страниц, считанных из кэша. Physical reads – число страниц, считанных с диска. Read-ahead reads обозначает число страниц, помещенных в кэш в ожидании будущего чтения.

Дополним наш анализ выполнением системной хранимой процедуры для получения статистических данных о размере таблицы.

```
sp_spaceused employees
```

Результаты:

```
name rows reserved data index_size unused
--------- ---- --------- ------- ----------- -------
Employees 2977 2008 KB 1504 KB 448 KB 56 KB
```

Что можно сказать, исходя из этой информации?

- Запрос не должен был сканировать всю таблицу. Количество данных в таблице составляет более 1,5 мегабайт, а потребовалось всего лишь 53 логические операции ввода-вывода для получения результата. Это говорит о том, что запрос обнаружил индекс, который мог быть использоваться для получения результата, а сканирование индекса потребляет меньше операций ввода-вывода, чем сканирование всех страниц с данными.

- Индексные страницы были, главным образом, обнаружены в кэше данных, поскольку значение physical reads равно 0. Этот запрос потому выполнился быстро, что другие запросы к таблице *employees*, выполненные прежде, уже кэшировали как таблицу, так и ее индекс. Скорость выполнения может измениться при других условиях.

- Microsoft не предоставил информации о какой-либо активности опережающего чтения. В данном случае страницы данных и индексные страницы уже находились в кэше. При сканировании большой таблицы опережающее чтение, вероятно, занесет в кэш необходимые страницы до того, как ваш запрос затребует их. Опережающее чтение включается автоматически, когда SQL Server определяет, что ваша транзакция считывает страницы из базы данных последовательно и может спрогнозировать, какая следующая страница потребуется. Фактически запускается отдельное соединение SQL Server, которое идет впереди вашего процесса и кэширует для него страницы данных. [Конфигурация и настройка параметров опережающего чтения выходит за пределы данной статьи.]

В нашем примере, запрос был выполнен наиболее эффективно и не нуждается в дальнейшей настройке.

SET STATISTICS TIME

Время выполнения транзакции является не постоянным критерием, т.к. он зависит от деятельности других пользователей сервера. Однако оно представляет некоторую реальную меру по сравнению с числом страниц данных, которые не имеют смысла для ваших пользователей. Их интересуют только секунды и минуты, которые они тратят на ожидание ответа на запрос, но не кэширование данных и не эффективность опережающего чтения. Команда *SET STATISTICS TIME ON* выдает реальное время выполнения и использование центрального процессора для каждого последующего запроса. Выполнение операции *SET STATISTICS TIME OFF* отменяет эту возможность.

```
SET STATISTICS TIME ON
GO
SELECT COUNT(*) FROM titleauthors
GO
SET STATISTICS TIME OFF
GO
```

Результаты:

```
SQL Server Execution Times:
 cpu time = 0 ms. elapsed time = 8672 ms.
SQL Server Parse and Compile Time:
 cpu time = 10 ms.

-----------
25

(1 row(s) affected)


SQL Server Execution Times:
 cpu time = 0 ms. elapsed time = 10 ms.
SQL Server Parse and Compile Time:
 cpu time = 0 ms.
```

Первое сообщение предоставляет несколько непонятное время выполнения – 8 672 миллисекунд. Это число не имеет отношение к нашему коду и показывает время, которое прошло с момента выполнения предыдущей команды. Этим сообщением можно пренебречь. Всего 10 миллисекунд потребовалось SQL Server для синтаксического разбора и компиляции запроса. А для его выполнения потребовалось 0 миллисекунд (показано после результата запроса). Фактически это говорит о том, что длительность запроса была слишком короткой для измерения. Последнее сообщение, показывающее время 0 мс на синтаксический разбор и компиляцию, относится к команде *STATISTICS TIME OFF* (т.е. столько потребовалось на ее компиляцию). Вы можете пренебречь и этим сообщением, т.к. более важные сообщение в выходных данных выделены.

Отметим также, что время выполнения и использования центрального процессора указывается в миллисекундах. Числа на вашем компьютере могут отличаться (не пытайтесь сравнивать характеристики вашего компьютера с нашим ноутбуком, потому что он не является эталоном). Более того, каждый раз при выполнении этого кода вы можете получать небольшие различия в получаемых статистических данных; это зависит от того, что еще выполняет в это время SQL Server.

Если вам необходимо измерить интервал времени на выполнение ряда запросов или хранимой процедуры, то более практично сделать это программно (будет описано ниже). Причина заключается в том, что *STATISTICS TIME* выводит длительность каждого отдельного запроса, и вам придется складывать эти данные вручную при запуске нескольких команд. Представьте размер выходных данных и количество ручной работы при оценке времени выполнения скрипта, который выполняет множество запросов в цикле тысячи раз!

Лучше рассмотрим следующий код, в котором снимаются показания времени до начала и после окончания транзакции, и выводится общая продолжительность выполнения в секундах (или, если хотите, в миллисекундах):

```
DECLARE @start_time DATETIME
SELECT @start_time = GETDATE()
< какой либо запрос или код, время которого необходимо вычислить, без
оператора GO >
SELECT 'Elapsed Time, sec' = DATEDIFF( second, @start_time, GETDATE() )
GO
```

Если ваш код содержит несколько блоков, выделенных оператором *GO*, то вы не можете использовать локальную переменную для сохранения времени начала выполнения. Переменная будет уничтожена после окончания блока, ограниченного командой *GO*, где она была создана. Однако вы можете сохранить время начала выполнения во временной таблице следующим образом:

```
CREATE TABLE #save_time ( start_time DATETIME NOT NULL )
INSERT #save_time VALUES ( GETDATE() )
GO
< какой-либо код, время выполнения которого необходимо измерить (может
включать GO) >
GO
SELECT 'Elapsed Time, sec' = DATEDIFF( second, start_time, GETDATE() )
FROM #save_time
DROP TABLE #save_time
GO
```

Необходимо помнить о том, что тип данных *DATETIME* в SQL Server сохраняет значения времени приращениями в 3 миллисекунды. Невозможно получить более точное значение времени чем то, которое использует тип данных *DATETIME*.

Выходные данные и анализ SHOWPLAN

В этой статье на примерах объяснения планов выполнения демонстрируется значение и полезность выходных данных, получаемых при использовании в Microsoft SQL Server 2000 команды SET SHOWPLAN TEXT ON. План объяснения (также называемый планом запроса, планом выполнения или планом оптимизации) предоставляет точную детализацию шагов, которые использует процессор запросов баз данных для выполнения SQL-транзакций. Умение читать планы выполнения повышает вашу способность профессионально настраивать и оптимизировать запросы.

Большинство примеров основаны либо на базе данных PUBS, либо на системных таблицах SQL Server. Для этих примеров мы добавили десятки тысяч строк во множество таблиц, чтобы оптимизатору запросов было с чем реально работать при оценке планов выполнения.

Выходные данные SHOWPLAN

Одной из причин, почему нам нравится оптимизатор запросов, является то, что он предоставляет обратную связь в виде плана выполнения запроса. В данной главе, мы рассмотрим это более детально и опишем сообщения, которые могут встретиться в планах выполнения. Понимание этих сообщений переводит ваши достижения в области оптимизации на новый уровень. Вы больше не будете рассматривать оптимизатор как «черный ящик», который касается ваших запросов волшебной палочкой.

Следующая команда предписывает SQL Server показать план выполнения для каждого запроса, который будет выполнен следом за этой командой в том же самом соединении, или отключает эту возможность.

SET SHOWPLAN_TEXT { ON | OFF }

По умолчанию, *SHOWPLAN_TEXT ON* приводит к тому, что код, который вы тестируете, не выполняется. Вместо этого, SQL Server компилирует код и выводит для него план выполнения запроса. Такое поведение продолжается до тех пор, пока не будет выполнена команда *SET SHOWPLAN_TEXT OFF*.

Другие полезные команды оператора SET

Здесь представлены команды оператора *SET*, которые полезны для настройки и отладки. Ранее в этом документе мы уже рассматривали *SET STATISTICS*. Здесь же представлены другие команды *SET*, которые могут быть полезны в определенных ситуациях:

1. *SET NOEXEC {ON|OFF}*: проверяет синтаксис вашего кода Transact-SQL, включая компиляцию кода, но без его выполнения. Это бывает полезным для проверки синтаксиса запроса, воспользовавшись преимуществом, которое дает разрешение еще неопределенных имен (deferred-name resolution). Т.е. для проверки синтаксиса запроса к таблице, которая еще не была создана.

2. *SET FMTONLY {ON|OFF}*: возвращает клиенту только метаданные запроса. Для операторов SELECT это обычно означает вернуть только заголовки столбцов.

3. *SET PARSEONLY {ON|OFF}*: обычно проверяет синтаксис вашего кода Transact-SQL, но без его компиляции и выполнения.

Все эти команды выполняются один раз оператором *ON* и действуют, пока явно не будут выключены (*OFF*). *И*х установка не приводит ни к какому результату, они начинают свою работу со следующего шага. Другими словами, вы должны использовать команду *GO*, прежде чем будут параметры *SHOWPLAN* или *NOEXEC* вступят в силу.

Типичный код T-SQL, который используется для получения плана выполнения запроса без фактического его выполнения, выглядит следующим образом:

```
SET SHOWPLAN_TEXT ON
GO
<запрос>
GO
SET SHOWPLAN_TEXT OFF
GO
```

Мы опишем выходные данные *SHOWPLAN_TEXT* на нескольких примерах. Для того чтобы сократить объем статьи, мы не будем повторять команды *SET*, описанные выше. Каждый пример этого раздела, представляет собой запрос, который должен быть подставлен вместо тега <запрос> данного скрипта и предполагает показанную выше «обертку».

Фактически существует две версии *SHOWPLAN*: *SHOWPLAN_ALL* и *SHOWPLAN_TEXT*. *И*нформация, получаемая в результате их выполнения по существу одна и та же. Однако результат работы *SHOWPLAN_ALL* предназначен для обработки запроса графическими средствами, а не для рассмотрения пользователем. *SHOWPLAN_TEXT*, который мы используем на протяжении всей нашей статьи, формирует выходные данные в более читабельном представлении. Следующий простой запрос выбирает все строки из таблицы *authors*. Здесь нет никакого другого выбора, кроме как сканировать всю таблицу, т.к. мы не использовали предложение WHERE:

```
SELECT * FROM authors
```

Результаты *SHOWPLAN_TEXT* не отформатированы, однако нам пришлось сильно ужать вывод команды *SHOWPLAN_ALL*, чтобы сделать его читабельным в нижеприведенной таблице.

SHOWPLAN_TEXT	SHOWPLAN_ALL
StmtText ---------- |--Clustered Index Scan (OBJECT:([pubs].[dbo]. [authors].[UPKCL_auidind]))	StmtText ---------- |--Clustered Index Scan (OBJECT:([pubs].[dbo]. [authors].[UPKCL_auidind])) StmtID NodeID Parent ------ ------ ------ 2 1 0 PhysicalOp LogicalOp ---------- ---------- NULL NULL Clustered Index Scan Clustered Index Scan Argument ---------- 1 OBJECT:([pubs].[dbo].[authors]. [UPKCL_auidin d]) DefinedValues ---------- 23 …<все столбцы таблицы>… EstimatedRows EstimatedIO EstimatedCPU ---------- ---------- ---------- 23 NULL NULL 23 0.01878925 5.1899999E-5 AvgRowSize TotalSubtreeCost ---------- ---------- NULL 3.7682299E-2 111 3.7682299E-2 OutputList ---------- NULL …<все столбцы таблицы>… Warnings Type Parallel EstimateExecutions -------- ------ -------- ---------- NULL SELECT 0 NULL NULL PLAN_ROW 0 1.0

В этом и заключаются их существенное различие. Оператор *SHOWPLAN_ALL* возвращает множество полезной для настройки информации, но она тяжела для понимания и применения.

Операции SHOWPLAN

Некоторые из операций *SHOWPLAN*, иногда называемые «тегами» очень понятно объясняют работу SQL Server, в то время как другие наоборот озадачивают. Все

эти операции делятся на физические и логические. Физические операции описывают физический алгоритм, используемый для выполнения запроса, например, осуществляющий поиск в индексе. Логические операции описывают такие операции реляционной алгебры, используемые оператором, как агрегация. Результаты *SHOWPLAN* делятся на шаги. Каждая физическая операция запроса представлена отдельным шагом. Обычно шаги включают логический оператор, но не все шаги содержат логические операции. Кроме того, большинство шагов содержат операцию (или логическую, или физическую) и аргумент. Аргументы представляют собой компонент запроса, на который воздействует операция.

Обсуждение всех шагов планов выполнения было бы недопустимо пространным. Поэтому вместо рассмотрения их здесь, вы можете обратиться к официальной статье “SHOWPLAN Output and Analysis” («Выходные данные и анализ SHOWPLAN») доступный на http://www.quest.com/whitepapers/#ms_sql_Server .

Чтение плана выполнения запроса

Вместо демонстрации примеров, включаемых в описание логических и физических операций, мы рассмотрим их по отдельности. Т.к. единственный пример может проиллюстрировать использование и эффективность сразу нескольких операторов.

Начинаем

Давайте начнем с нескольких простых примеров, которые помогут нам понять, как следует читать план выполнения запроса, который выдается либо при вызове команды *SET SHOWPLAN_TEXT ON*, либо при установке опции с аналогичным именем в конфигурационных свойствах SQL Query Analyzer.

Этот пример использует *pubs..big_sales*, точную копию таблицы *pubs..sales*, только содержащую 80,000 записей, в качестве главного источника примеров простых планов выполнения.

Простейший запрос, такой как рассмотренный выше, будет сканировать весь кластеризованный индекс, если он существует. Помните, что кластеризованный ключ представляет физический порядок, в котором записываются данные. Следовательно, если кластеризованный ключ существует, вы можете избежать сканирования таблицы. Даже если вы выбираете столбец, который, в частности, не определен в кластеризованном ключе, например, ord_date, оптимизатор запросов будет использовать сканирование кластеризованного индекса, для получения результирующего набора.

```
SELECT *
FROM big_sales

SELECT ord_date
FROM big_sales

StmtText
------------------------------------------------------------------------------
|--Clustered Index Scan(OBJECT:([pubs].[dbo].[big_sales].[UPKCL_big_sales]))
```

Запросы, показанные выше, возвращают сильно различающиеся по количеству наборы данных, поэтому запрос, который вернет меньше данных (ord_date) выполнится быстрее, чем другой запрос просто потому, что потребует намного меньше операций ввода-вывода. Однако, планы выполнения запросов фактически идентичны.

Вы можете улучшить производительность, с помощью использования альтернативных индексов. Например, на столбце title_id имеется некластеризованный индекс:

```
SELECT title_id
FROM big_sales


StmtText
------------------------------------------------------------------------
|--Index Scan(OBJECT:([pubs].[dbo].[big_sales].[ndx_sales_ttlID]))
```

Этот запрос выполняется быстрее, чем запрос SELECT *, т.к. он может полностью получить необходимый ответ из некластеризованного индекса. Этот тип запроса называется «покрывающим запросом», т.к. весь результирующий набор входит в некластеризованный индекс.

Сравнение SEEK и SCAN

Первое, что вам необходимо будет уяснить в плане выполнения запросов, это различие между операциями *SEEK* и *SCAN*.

Не сложное, но очень полезное эмпирическое правило гласит, что операции *SEEK* хороши, операции *SCAN* – не очень, если не сказать плохи. Поиск напрямую, или, по крайней мере, очень быстро, обращается к нужным записям, тогда как сканирование считывает весь объект (таблицу, кластеризованный индекс или некластеризованный индекс). Таким образом, сканирование обычно потребляет больше ресурсов, чем поиск.

Если план выполнения запроса показывает только сканирование, то необходимо подумать о настройке запроса.

Предложение *WHERE* может значительно изменить производительность запроса, что показано ниже.

```
SELECT *
FROM big_sales
WHERE stor_id = '6380'

StmtText
------------------------------------------------------------------------------
|--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].[UPKCL_big_sales]),
SEEK:([big_sales].[stor_id]=[@1]) ORDERED FORWARD)
```

Представленный выше запрос теперь может использовать операцию *SEEK*, а не *SCAN* на кластеризованном индексе. *SHOWPLAN* явно демонстрирует нам, что операция поиска основана на (*stor_id*), и что результаты *УПОРЯДОЧЕНЫ* в соответствии с их текущим размещением в указанном индексе. Начиная с SQL Server 2000, поддерживается как прямое, так и обратное направление обхода индексов с одинаковой эффективностью, и вы можете увидеть в плане выполнения запроса *ORDERED FORWARD* или *ORDERED BACKWARD*. Именно это сообщает вам, в каком направлении происходило чтение таблицы или индекса. Вы даже можете управлять направлением обхода при помощи служебных слов *ASC* и *DESC* в предложении *ORDER BY*.

Запросы по диапазону возвращают планы выполнения, которые выглядят очень похоже на прямой запрос, показанный выше. Следующие два запроса пояснят сказанное.

```
SELECT *
FROM big_sales
WHERE stor_id >= '7131'

StmtText
------------------------------------------------------------------------------
|--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].[UPKCL_big_sales]),
SEEK:([big_sales].[stor_id] >= '7131') ORDERED FORWARD)
```

Вышеприведенный запрос выглядит почти также как и предыдущий, однако, предикат *SEEK* несколько другой.

```
SELECT *
FROM big_sales
WHERE stor_id BETWEEN '7066' AND '7131'

StmtText
------------------------------------------------------------------------
|--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].[UPKCL_big_sales]),
SEEK:([big_sales].[stor_id] >= '7066' AND [big_sales].[stor_id] <= '7131')
ORDERED FORWARD)
```

Этот запрос выглядит примерно так же, однако предикат *SEEK* изменился. Поскольку *SEEK* является очень быстрой операцией, то данный запрос достаточно хороший.

Операции *SEEK* и *SCAN* могут включать предикат предложения *WHERE*. В этом случае предикат скажет вам, какие записи из результирующего набора отфильтрует предложение *WHERE*. Поскольку предикат *WHERE* является компонентом операции *SEEK* или *SCAN*, то он сам по себе зачастую ни улучшает, ни ухудшает производительность этих операций. Однако предложение *WHERE* помогает оптимизатору запросов найти наилучшим образом подходящие индексы для обработки запроса.

Важная часть настройки запроса – выяснение, выполняется ли операция *SEEK* с использованием каких-либо индексов, и если да, то какие из этих индексов являются наилучшими. Большую часть времени оптимизатор запросов будет тратить на поиск существующих индексов. Однако с индексами связаны три общих проблемы:

1. Разработчик базы данных, чаще разработчик приложения, не создавал никаких индексов на таблицах.

2. Разработчик базы данных не предугадал типы запросов или транзакций, которые по большей части будут выполняться для данных таблиц, поэтому некоторые индексы или первичные ключи для данных таблиц оказываются неэффективными.

3. Разработчик базы данных угадал с индексированием таблиц при их создании, но транзакционная нагрузка со временем изменилась, и индексы стали менее эффективными.

Если вы видите в плане выполнения запроса множество операций сканирования и намного меньше операций поиска, то необходимо пересмотреть используемые индексы. Например, рассмотрим нижеследующий пример:

```
SELECT ord_num
FROM sales
WHERE ord_date IS NOT NULL
AND ord_date > 'Jan 01, 2002 12:00:00 AM'
```

```
StmtText
------------------------------------------------------------------------------
|--Clustered Index Scan(OBJECT:([pubs].[dbo].[sales].[UPKCL_sales]),
WHERE:([sales].[ord_date]>'Jan 1 2002 12:00AM'))
```

Этот запрос содержит предложение *WHERE* для столбца *ord_date*, хотя операции поиска по индексу. Если посмотреть на таблицу, то можно увидеть, что столбец *ord_date* не имеет индекса, но он по всей вероятности необходим. Если его добавить, то план выполнения запроса будет выглядеть следующим образом:

```
StmtText
------------------------------------------------------------------------------
|--Index Seek(OBJECT:([pubs].[dbo].[sales].[sales_ord_date]),
SEEK:([sales].[ord_date] > 'Jan 1 2002 12:00AM') ORDERED FORWARD)
```

Теперь запрос использует операцию *INDEX SEEK* (поиск по индексу) для созданного нами индекса sales_ord_date.

Ветвящиеся шаги, демонстрируемые сравнением соединений и подзапросов

Старое эмпирическое правило гласит, что соединения предпочтительнее подзапросов, дающих такой же результирующий набор.

```
SELECT au_fname, au_lname
FROM authors
WHERE au_id IN
(SELECT au_id FROM titleauthor)

StmtText
------------------------------------------------------------------------------
|--Nested Loops(Inner Join, OUTER REFERENCES:([titleauthor].[au_id]))
   |--Stream Aggregate(GROUP BY:([titleauthor].[au_id]))
   |  |--Index Scan(OBJECT:([pubs].[dbo].[titleauthor].[auidind]), ORDERED
        FORWARD)
   |--Clustered Index Seek(OBJECT:([pubs].[dbo].[authors].[UPKCL_auidind]),
    SEEK:([authors].[au_id]=[titleauthor].[au_id]) ORDERED FORWARD)

Table 'authors'. Scan count 38, logical reads 76, physical reads 0, read-
ahead reads 0.
Table 'titleauthor'. Scan count 2, logical reads 2, physical reads 1, read-
ahead reads 0.
```

В этом случае оптимизатор запросов выбирает операцию с вложенным циклом (*nested loop*). Запрос вынужден считать всю таблицу *authors*, используя поиск в кластеризованном индексе и ограничиваясь при этом лишь чтением логических страниц.

В запросах с ветвлением, прямыми линиями с отступами показаны шаги, которые являются ветвями других шагов.

Теперь рассмотрим соединение:

```
SELECT DISTINCT au_fname, au_lname
FROM authors AS a
JOIN titleauthor AS t ON a.au_id = t.au_id
```

```
StmtText
------------------------------------------------------------------------------
|--Stream Aggregate(GROUP BY:([a].[au_lname], [a].[au_fname]))
   |--Nested Loops(Inner Join, OUTER REFERENCES:([a].[au_id]))
      |--Index Scan(OBJECT:([pubs].[dbo].[authors].[aunmind] AS [a]), ORDERED
         FORWARD)
      |--Index Seek(OBJECT:([pubs].[dbo].[titleauthor].[auidind] AS [t]),
       SEEK:([t].[au_id]=[a].[au_id]) ORDERED FORWARD)

Table 'titleauthor'. Scan count 23, logical reads 23, physical reads 0,
readahead reads 0.
Table 'authors'. Scan count 1, logical reads 1, physical reads 0, read-ahead
reads 0.
```

Для приведенного запроса число операций логического чтения для таблицы *titleauthors* увеличивается, а для таблицы *authors* – уменьшается. Следует обратить внимание, что соединение данных происходит выше (позже) в плане выполнения запроса.

Сравнение планов выполнения запросов

Будем использовать планы выполнения запросов для сравнения относительной эффективности двух отдельных запросов. Например, нужно посмотреть тратит ли один запрос больше ресурсов, чем другой, или использует ли он другую индексную стратегию.

В данном примере, сравним два запроса. Первый использует *SUBSTRING*, а второй – *LIKE*:

```
SELECT *
FROM authors
WHERE SUBSTRING( au_lname, 1, 2 )= 'Wh'

StmtText
------------------------------------------------------------------------------
|--Clustered Index Scan(OBJECT:([pubs].[dbo].[authors].[UPKCL_auidind]),
  WHERE:(substring([authors].[au_lname], 1, 2)='Wh'))
```

Сравним его с аналогичным запросом, использующим *LIKE*:

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

StmtText
------------------------------------------------------------------------------
|--Bookmark Lookup(BOOKMARK:([Bmk1000]), OBJECT:([pubs].[dbo].[authors]))
   |--Index Seek(OBJECT:([pubs].[dbo].[authors].[aunmind]),
    SEEK:([authors].[au_lname] >= 'WGþ' AND [authors].[au_lname] < 'WI'),
    WHERE:(like([authors].[au_lname], 'Wh%', NULL)) ORDERED FORWARD)
```

Очевидно, что второй запрос, использующий операцию *INDEX SEEK*, имеет более простой план выполнения запроса, чем первый с его *CLUSTERED INDEX SCAN*.

Для определения лучшего планы выполнения запросов предпочтительней использовать либо *SET STATISTICS PROFILE ON*, либо опцию SQL Query Analyzer *Graphic Execution Plan*, а не *SET SHOWPLAN_TEXT ON*. Эти средства точно покажут вам, какую часть обработки данных, в процентах, потребляет каждый шаг в

плане выполнения запроса. Это дает возможность сказать, какой вариант является более или менее затратным для оптимизатора запросов. Также можно сразу выполнить два (или более) запроса и сравнить, какой из них является более эффективным.

Также важно использовать *SET STATISTICS IO* и *SET STATISTICS TIME* для наиболее полной оценки возможной производительности.

Объяснение воздействия соединений

Если вы прочитали выше все различные шаги выполнения запросов, то заметили, какое большое количество операций было привлечено для объяснения происходящего с соединениями в SQL Server 2000. Каждая стратегия соединения имеется как свои недостатки, так и свои достоинства. Однако изредка возникают ситуации, когда оптимизатор запросов выбирает менее эффективное соединение, обычно используя стратегию хеширования или слияния, в то время как простой вложенный цикл оказывается более эффективным.

SQL Server применяет три стратегии соединения. Они перечислены ниже по степени возрастания сложности.

Nested Loop (Вложенный Цикл)

Является наилучшей стратегией для маленьких таблиц с простыми внутренними соединениями. Лучше всего она работает в случае, когда одна таблица имеет сравнительно меньшее число записей по сравнению с другой таблицей, И обе они проиндексированы по соединяемым столбцам. Соединения с помощью вложенного цикла (*Nested loop*) требуют наименьшего числа операций ввода-вывода и наименьшего количества сравнений.

Цикл *Nested loop* проходит один раз все записи во внешней таблице (желательно, в меньшей таблице), на каждом шаге которого выполняется поиск совпадений во внутренней таблице и формируется выходной набор. Существует много названий для конкретных стратегий вложенных циклов. Например, соединение с помощью примитивного вложенного цикла (*naive nested loop join*) относится к случаю, когда происходит поиск по всей таблице или индексу. Соединением с помощью индексного вложенного цикла (*index nested loop join*) или соединением с помощью временного индексного вложенного цикла (*temporary index nested loop join*) называется соединение, когда используется постоянный или временный индекс.

Merge (Слияние)

Наилучшая стратегия для больших таблиц приблизительно одинаковых по размеру и отсортированных по соединяемым столбцам. Операции слияния (*Merge*) сортируют и затем циклически обходят все данные для формирования выходного потока. Преимущество операции объединения слиянием основано на наличии индексов на соответствующем наборе столбцов, почти всегда это столбцы, указанные в предложении равенства для предиката соединения.

Соединения *Merge* имеют преимущества для предварительно отсортированных наборов, выбирая по строке из каждого входного набора и выполняя прямое сравнение. Например, внутренние соединения возвращают записи, когда предикаты соединения есть равенство. Если они не равны, то запись с меньшим значением отбрасывается, и для сравнения выбирается следующая запись. Этот процесс продолжается до тех пор, пока не будут проверены все записи. Иногда соединения слиянием используются для сравнения таблиц в отношениях «многие-ко-многим». В этом случае SQL Server использует временную таблицу для хранения строк. Если в запросе с соединением merge также существует предложение *WHERE*, то предикат *join merge* оценивается первым. Затем записи, полученные после выполнения предиката *merge join*, проверяются на удовлетворение другим предикатам в предложении *WHERE*. Microsoft называет такие предикаты остаточными.

Хеш (Hash)

Hashes join является лучшей стратегией для больших таблиц разного размера или для сложных условий соединения, когда соединяемые столбцы не индексированы или отсортированы. Хеширование используется для операций *UNION, INTERSECT, INNER, LEFT, RIGHT и FULL OUTER JOIN*, а также для множественных операций соответствия и расхождения. Хеширование также используется для соединений таблиц, когда не существует полезных индексов. Операции хеширования создают временную хеш-таблицу и, затем, циклически обходит все данные, формируя выходной поток.

Hash использует входной поток создания хеш-таблицы (*build input*) (всегда меньшая таблица) и входной поток зондирования (*probe input*). Для формирования соединения запрос использует хеш-ключ (т.е. столбцы, указанные в предикате соединения (*join*) или иногда перечисленные в списке *GROUP BY*). Остаточными предикатами в этом случае являются любые критерии, указанные в предложении *WHERE*, которые не используются в самом предикате соединения. Остаточные предикаты проверяются после предикатов соединения. Существует несколько различных опций, которые SQL Server выбирает при формировании *hash join*. Они указаны в порядке старшинства:

In-memory Hash: Соединения *in-memory hash* создают в оперативной памяти временную хеш-таблицу при первом сканировании входного потока создания в память. Каждая запись вставляется в хеш-сегмент (*hash bucket*), определяемый хеш-значением, вычисленном для хеш-ключа. Далее входной поток зондирования сканируется запись за записью. Каждое значение входного потока зондирования сравнивается с соответствующим хеш-сегментом, и при совпадении заносится в результирующий набор.

Hybrid Hash (гибридное хеширование): Если размер хеш-таблицы незначительно превышает объем доступной памяти, SQL Server может использовать комбинацию методов *in-memory hash join* и *grace hash join*, это и будет называется соединением *hybrid hash join*.

Grace Hash: Способ *grace hash* используется, когда соединение хешированием дает очень большой входной поток создания, который не может

быть обработан в оперативной памяти. В этом случае полностью считываются входные потоки создания и зондирования. Затем они помещаются во множественные временные рабочие таблицы. Этот шаг называется секционированием (*partitioning fan-out*). Хеш-функция для хеш-ключей гарантирует, что все соединяемые записи находятся в одной и той же паре секционированных рабочих таблиц. Секционирование разбивает два больших шага на множество мелких шагов, которые могут обрабатываться параллельно. Хеш-соединение теперь применяется для каждой пары рабочих таблиц, и любые совпадения попадают в результирующий набор.

Recursive Hash (рекурсивное хеширование): Иногда секционированные таблицы, применяемые в *Grace hash*, остаются все еще достаточно большими и требуют дальнейшего секционирования. Это явление носит название *recursive hash*. Необходимо заметить что *hash*- и *merge*-соединения проходят каждую таблицу один раз. Поэтому они могут дать обманчиво невысокие показатели по числу операций ввода-вывода, если вы будете использовать команду *SET STATISTICS IO ON* с запросами этого типа. Тем не менее, низкий показатель операций ввода-вывода не означает, что эти стратегии соединения по определению быстрее, чем соединения с вложенными циклами, из-за их огромных вычислительных затрат.

Хеш-соединения, в частности, являются дорогими по отношению к вычислениям. Если у вас в действующем приложении есть определенные запросы, последовательно использующие хеш-соединения, то это повод для настройки запроса или добавления индексов на используемые таблицы.

В следующем примере показаны как стандартный вложенный цикл (*nested loop*) (использующий план выполнения запроса по умолчанию), так и *hash*- и *merge*-соединения (принудительное использование которых обеспечивается указаниями оптимизатору (*hint*)).

```
SELECT a.au_fname, a.au_lname, t.title
FROM authors AS a
INNER JOIN titleauthor ta
ON a.au_id = ta.au_id
INNER JOIN titles t
ON t.title_id = ta.title_id
ORDER BY au_lname ASC, au_fname ASC

StmtText
------------------------------------------------------------------------------
 |--Nested Loops(Inner Join, OUTER REFERENCES:([ta].[title_id]))
    |--Nested Loops(Inner Join, OUTER REFERENCES:([a].[au_id]))
    |  |--Index Scan(OBJECT:([pubs].[dbo].[authors].[aunmind] AS [a]),
          ORDERED FORWARD)
    |  |--Index Seek(OBJECT:([pubs].[dbo].[titleauthor].[auidind] AS [ta]),
          SEEK:([ta].[au_id]=[a].[au_id]) ORDERED FORWARD)
    |--Clustered Index Seek(OBJECT:([pubs].[dbo].[titles].[UPKCL_titleidind]
       AS [t]), SEEK:([t].[title_id]=[ta].[title_id]) ORDERED FORWARD)
```

План, показанный выше, является стандартным планом выполнения запроса, который создает SQL Server. Можно потребовать с помощью указаний, чтобы SQL Server показал, как бы он обрабатывал *merge*- и *hash*-соединения.

```
SELECT a.au_fname, a.au_lname, t.title
FROM authors AS a
INNER MERGE JOIN titleauthor ta
ON a.au_id = ta.au_id
INNER HASH JOIN titles t
ON t.title_id = ta.title_id
ORDER BY au_lname ASC, au_fname ASC

Warning: The join order has been enforced because a local join hint is used.

StmtText
------------------------------------------------------------------------------
 |--Sort(ORDER BY:([a].[au_lname] ASC, [a].[au_fname] ASC))
    |--Hash Match(Inner Join, HASH:([ta].[title_id])=([t].[title_id]),
     RESIDUAL:([ta].[title_id]=[t].[title_id]))
       |--Merge Join(Inner Join, MERGE:([a].[au_id])=([ta].[au_id]),
         RESIDUAL:([ta].[au_id]=[a].[au_id]))
       |  |--Clustered Index
Scan(OBJECT:([pubs].[dbo].[authors].[UPKCL_auidind]
           AS [a]), ORDERED FORWARD)
       |  |--Index Scan(OBJECT:([pubs].[dbo].[titleauthor].[auidind] AS [ta]),
            ORDERED FORWARD)
       |--Index Scan(OBJECT:([pubs].[dbo].[titles].[titleind] AS [t]))
```

В данном примере ясно видно, что каждое из соединений рассматривает предикат другого соединения как остаточный предикат. (Также можно заметить, что при использовании указаний (*hint*) SQL Server выдает предупреждение.) Этот запрос также вынужден использовать оператор SORT, чтобы могли быть использованы *hash*- и *merge*-соединения.

Методы настройки запросов

В данной статье на примерах и планах выполнения показаны полезные технические приемы для улучшения запросов в Microsoft SQL Server 2000. Существует множество небольших подсказок и технических приемов, применимых для небольшого класса задач программирования. Их знание расширит ваши возможности в оптимизации производительности. Для отображения выходных данных всех примеров в этом разделе используется *SHOWPLAN_ALL*, т.к. его вывод более компактен и при этом показываются вся критически важная информация. Большинство примеров основано либо на базе данных *PUBS*, либо на системных таблицах. Мы значительно увеличили размер таблиц, используемых в базе данных *PUBS*, добавив десятки тысяч записей во многие из них.

Оптимизация подзапросов

В качестве хорошего эмпирического правила, попытайтесь заменить все подзапросы соединениями. Иногда оптимизатор может автоматически выравнивать подзапросы, заменяя их обычными или внешними соединениями. Но это не всегда дает хороший результат. Явно указанные соединения дают оптимизатору больше возможности для выбора порядка использования таблиц и нахождения наилучшего из всех возможных планов. При оптимизации конкретного запроса необходимо проанализировать изменяется ли каким-нибудь образом план при замене подзапросов.

Пример

В следующих запросах необходимо получить названия магазинов и количество книг, проданных в каждом магазине. Оба запроса возвращают один и тот же результирующий набор, но первый из них использует подзапрос, а второй – внешнее соединение. Сравним планы выполнения запросов, произведенные Microsoft SQL Server.

Решение с помощью подзапроса	Решение с помощью соединения
SELECT st.stor_name AS 'Store', (SELECT SUM(bs.qty) FROM big_sales AS bs WHERE bs.stor_id = st.stor_id), 0) AS 'Books Sold' FROM stores AS st WHERE st.stor_id IN (SELECT DISTINCT stor_id FROM big_sales)	SELECT st.stor_name AS 'Store', SUM(bs.qty) AS 'Books Sold' FROM stores AS st JOIN big_sales AS bs ON bs.stor_id = st.stor_id WHERE st.stor_id IN (SELECT DISTINCT stor_id FROM big_sales) GROUP BY st.stor_name
SQL Server parse and compile time: CPU time = 28 ms, elapsed time = 28 ms. SQL Server Execution Times: CPU time = 145 ms, elapsed time = 145 ms.	SQL Server parse and compile time: CPU time = 50 ms, elapsed time = 54 ms. SQL Server Execution Times: CPU time = 109 ms, elapsed time = 109 ms.
Table 'big_sales'. Scan count 14, logical reads 1884, physical reads 0, read-ahead reads 0. Table 'stores'. Scan count 12, logical	Table 'big_sales'. Scan count 14, logical reads 966, physical reads 0, read-ahead reads 0. Table 'stores'. Scan count 12, logical

Решение с помощью подзапроса	Решение с помощью соединения
reads 24, physical reads 0, read-ahead reads 0.	reads 24, physical reads 0, read-ahead reads 0.

Глубоко не вникая, видно, что соединение работает быстрее. На это указывает и время работы центрального процессора, и фактическое время выполнения, используя почти вдвое меньше операций логического чтения, чем решение с помощью подзапроса.

Заметим, что результирующие множества совпадают в обоих случаях, хотя порядок сортировки различен, поскольку запрос с соединением (с его предложением *GROUP BY*) подразумевает неявное использование *ORDER BY*:

```
Store                Books Sold
------------------------------------ -----------
Barnum's             154125
Bookbeat             518080
Doc-U-Mat: Quality Laundry and Books 581130
Eric the Read Books  76931
Fricative Bookshop   259060
News & Brews         161090

(6 row(s) affected)

Store                Books Sold
------------------------------------ -----------
Eric the Read Books  76931
Barnum's             154125
News & Brews         161090
Doc-U-Mat: Quality Laundry and Books 581130
Fricative Bookshop   259060
Bookbeat             518080

(6 row(s) affected)
```

Изучение плана выполнения запроса для метода с подзапросом показывает:

```
|--Compute Scalar(DEFINE:([Expr1006]=isnull([Expr1004], 0)))
   |--Nested Loops(Left Outer Join, OUTER REFERENCES:([st].[stor_id]))
      |--Nested Loops(Inner Join, OUTER REFERENCES:([big_sales].[stor_id]))
      |  |--Stream Aggregate(GROUP BY:([big_sales].[stor_id]))
      |  |  |--Clustered Index Scan(OBJECT:([pubs].[dbo].[big_sales].
               [UPKCL_big_sales]), ORDERED FORWARD)
      |  |--Clustered Index Seek(OBJECT:([pubs].[dbo].[stores].[UPK_storeid]
            AS [st]),
          SEEK:([st].[stor_id]=[big_sales].[stor_id]) ORDERED FORWARD)
      |--Stream Aggregate(DEFINE:([Expr1004]=SUM([bs].[qty])))
         |--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].
            [UPKCL_big_sales] AS [bs]),
            SEEK:([bs].[stor_id]=[st].[stor_id]) ORDERED FORWARD)
```

Тогда как с запросом, использующим соединение, имеем:

```
 |--Stream Aggregate(GROUP BY:([st].[stor_name])
DEFINE:([Expr1004]=SUM([partialagg1005])))
    |--Sort(ORDER BY:([st].[stor_name] ASC))
       |--Nested Loops(Left Semi Join, OUTER REFERENCES:([st].[stor_id]))
          |--Nested Loops(Inner Join, OUTER REFERENCES:([bs].[stor_id]))
          |  |--Stream Aggregate(GROUP BY:([bs].[stor_id])
           DEFINE:([partialagg1005]=SUM([bs].[qty])))
          |  |  |--Clustered Index Scan(OBJECT:([pubs].[dbo].[big_sales].
                [UPKCL_big_sales] AS [bs]), ORDERED FORWARD)
          |  |--Clustered Index Seek(OBJECT:([pubs].[dbo].[stores].
              [UPK_storeid] AS [st]),
              SEEK:([st].[stor_id]=[bs].[stor_id]) ORDERED FORWARD)
          |--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].
           [UPKCL_big_sales]),
           SEEK:([big_sales].[stor_id]=[st].[stor_id]) ORDERED FORWARD)
```

Вариант решения с помощью соединения является более эффективным. Он не требует дополнительного агрегирующего потока, который производит суммирование по столбцу *big_sales.qty*, необходимое для обработки подзапроса.

UNION против UNION ALL

В любом случае, когда возможно используйте *UNION ALL* вместо *UNION*. Различие между ними заключается в том, что *UNION* имеет «побочный эффект» в виде устранения всех дубликатов и сортировку результатов, в то время как *UNION ALL* этого не делает. Для выбора неповторяющихся данных требуется создание временной рабочей таблицы, в которой хранятся все строки и происходит сортировка перед выводом результата. (Вывод плана выполнения для запроса *SELECT DISTINCT* покажет, что имеет место агрегация потока, потребляющая 30% всех ресурсов, используемых для выполнения запроса.) В некоторых случаях, это именно то, что вам необходимо – и тогда *UNION* ваш друг. Но если в результирующем множестве не предполагается наличие дублированных строк, то лучше использовать *UNION ALL*. Он просто осуществляет выборку из одной таблицы или соединения, а затем из другой, добавляя результаты к концу первого результирующего набора. *UNION ALL* не требует никакой временной рабочей таблицы и никакой сортировки (если нет других условий, требующих сортировки). В большинстве случаев это намного более эффективно. Существует еще одна потенциальная проблема – опасность переполнения базы данных *tempdb* огромной вспомогательной рабочей таблицей. Это может произойти в том случае, если запрос с *UNION* должен выдать большой результирующий набор.

Пример

Следующие запросы выбирают ID для всех магазинов в таблице *sales*, которая представляет собой оригинал таблицы в базе данных pubs, и все ID для магазинов из таблицы *big_sales* - версии таблицы *sales*, в которую мы добавили приблизительно 70 000 строк. Различие между двумя решениями заключается только в использовании *UNION* вместо *UNION ALL*. Однако добавление ключевого слова *ALL* существенно влияет на план выполнения запроса. Первое решение требует агрегации потоков и сортировку результатов, прежде чем они будут возвращены клиенту. Второй запрос намного более эффективен, особенно для больших таблиц. В этом примере оба запроса возвращают одинаковые результирующие наборы, хотя и в разном порядке. В наших тестах мы использовали использовали две временные таблицы во время выполнения. Поэтому ваши результаты могут отличаться.

Решение с UNION	Решение с UNION ALL
SELECT stor_id FROM big_sales UNION SELECT stor_id FROM sales	SELECT stor_id FROM big_sales UNION ALL SELECT stor_id FROM sales
|--Merge Join(Union) |--Stream Aggregate(GROUP BY: ([big_sales].[stor_id])) | |--Clustered Index Scan (OBJECT:([pubs].[dbo]. [big_sales]. [UPKCL_big_sales]), ORDERED FORWARD) |--Stream Aggregate(GROUP BY: ([sales].[stor_id])) |--Clustered Index Scan (OBJECT:([pubs].[dbo]. [sales].[UPKCL_sales]), ORDERED FORWARD)	|--Concatenation |--Index Scan (OBJECT:([pubs].[dbo]. [big_sales].[ndx_sales_ttlID])) |--Index Scan (OBJECT:([pubs].[dbo]. [sales].[titleidind]))
Table 'sales'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0. Table 'big_sales'. Scan count 1, logical reads 463, physical reads 0, read-ahead reads 0.	Table 'sales'. Scan count 1, logical reads 1, physical reads 0, read-ahead reads 0. Table 'big_sales'. Scan count 1, logical reads 224, physical reads 0, read-ahead reads 0.

Хотя результирующие наборы в этом примере аналогичны, можно увидеть, что предложение *UNION ALL* потребовало почти вдвое меньше ресурсов по сравнению с теми, которые потребовались для операции *UNION*. Так что если существует уверенность в том, что в ожидаемом результирующем наборе нет дубликатов, используйте предложение *UNION ALL*.

Функции и выражения, подавляющие использование индексов

Если применяются встроенные функции или выражения для индексированных столбцов, то оптимизатор не может использовать индексы на этих столбцах. Можно попытаться перезаписать эти условия таким образом, чтобы индексные ключи не включались ни в одно из выражений.

Примеры

Необходимо помогать SQL Server удалять любые выражения, применяемые к числовым столбцам, которые формируют индекс. Следующие запросы выбирают строку из таблицы jobs по уникальному ключу, который имеет уникальный кластеризованный индекс. Если применять выражение к столбцу, то индекс использоваться не будет. Но если только изменить условие ‘job_id – 2 = 0’ на ‘job_id=2’ оптимизатор выполнит операцию поиска (seek) по кластеризованному индексу.

Запрос, с неиспользующимся индексом	Оптимизированный запрос, использующий индекс

`SELECT *`
`FROM jobs`
`WHERE (job_id-2) = 0`
	`SELECT *`
	`FROM jobs`
	`WHERE job_id = 2`

`|--Clustered Index Scan(OBJECT:`
`([pubs].[dbo].[jobs].`
`[PK__jobs__117F9D94]),`
`WHERE:(Convert([jobs].[job_id])-`
`2=0))`
	`|--Clustered Index Seek(OBJECT:`
	`([pubs].[dbo].[jobs].`
	`[PK__jobs__117F9D94]),`
	
	`SEEK:([jobs].[job_id]=Convert([@1]))`
	`ORDERED FORWARD)`
	
	Заметим, что поиск (*seek*) значительно лучше сканирования (*scan*), как в предыдущем запросе.

В следующей таблице содержится больше примеров запросов, которые подавляют использование индекса на столбцах различного типа, а также способы их оптимизации.

Запрос, с не использующимся индексом	Оптимизированный запрос, использующий индекс

`DECLARE @job_id VARCHAR(5)`
`SELECT @job_id = '2'`
`SELECT *`
`FROM jobs`
`WHERE CONVERT( VARCHAR(5),`
`job_id ) = @job_id`
	`DECLARE @job_id VARCHAR(5)`
	`SELECT @job_id = '2'`
	`SELECT *`
	`FROM jobs`
	`WHERE job_id = CONVERT(`
	`SMALLINT, @job_id )`

`SELECT *`
`FROM authors`
`WHERE au_fname + ' ' + au_lname`
`= 'Johnson White'`
	`SELECT *`
	`FROM authors`
	`WHERE au_fname = 'Johnson'`
	`AND au_lname = 'White'`

`SELECT *`
`FROM authors`
`WHERE SUBSTRING( au_lname, 1, 2 ) =`
`'Wh'`
	`SELECT *`
	`FROM authors`
	`WHERE au_lname LIKE 'Wh%'`

`CREATE INDEX employee_hire_date`
`ON employee ( hire_date )`
`GO`
`-- Получаем всех сотрудников, нанятых`
`-- в первом квартале 1990 года:`
`SELECT *`
`FROM employee`
`WHERE DATEPART( year, hire_date ) =`
`1990`
	`CREATE INDEX employee_hire_date`
	`ON employee ( hire_date )`
	`GO`
	`-- Получаем всех сотрудников, нанятых`
	`-- в первом квартале 1990 года:`
	`SELECT *`
	`FROM employee`
	`WHERE hire_date >= '1/1/1990'`
	`AND hire_date < '4/1/1990'`

Запрос, с не использующимся индексом	Оптимизированный запрос, использующий индекс
AND DATEPART(quarter, hire_date) = 1	
-- Предположим, что дата найма -- может содержать время, отличное от 12AM -- Кто был нанят 2/21/1990? SELECT * FROM employee WHERE CONVERT(CHAR(10), hire_date, 101) = '2/21/1990'	-- Предположим, что дата найма -- может содержать время, отличное от 12AM -- Кто был нанят 2/21/1990? SELECT * FROM employee WHERE hire_date >= '2/21/1990' AND hire_date < '2/22/1990'

SET NOCOUNT ON

Эффект увеличения быстродействия кода T-SQL при использовании *SET NOCOUNT ON* остается малопонятным для многих разработчиков SQL Server и администраторов баз данных. Возможно, вы обращали внимание, что запросы, выполнившиеся успешно, возвращают системное сообщение о количестве обработанных ими строк. Во многих случаях эта информация не нужна. Команда *SET NOCOUNT ON* позволяет запретить вывод сообщений для всех последующих транзакций в этой сессии до тех, пока не будет выполнена команда *SET NOCOUNT OFF*. Эта опция оказывает не только косметический эффект на выходную информацию, генерируемую скриптом. Она уменьшает количество передаваемой от сервера клиенту информации. Поэтому эта команда позволяет снизить сетевой трафик и уменьшить общее время ответа транзакций. Время для передачи одного сообщения может быть и незначительным, но подумайте о скрипте, обрабатывающим в цикле несколько запросов и передающей килобайты бесполезной для пользователя информации.

В качестве примера рассмотрим файл, содержащий пакет T-SQL, который вставляет в таблицу *big_sales* 9999 строк.

```
-- Предполагается наличие таблицы BIG_SALES, которая является копией таблицы
pubs..sales

SET NOCOUNT ON

DECLARE @separator VARCHAR(25),
        @message VARCHAR(25),
        @counter INT,
        @ord_nbr VARCHAR(20),
        @order_date DATETIME,
        @qty_sold INT,
        @terms VARCHAR(12),
        @title CHAR(6),
        @starttime DATETIME

SET @STARTTIME = GETDATE()

SELECT @counter = 0,
       @separator = REPLICATE( '-', 25 )
       WHILE @counter < 9999
```

```
BEGIN
SET @counter = @counter + 1
SET @ord_nbr = 'Y' + CAST(@counter AS VARCHAR(5))
SET @order_date = DATEADD(hour, (@counter * 8), 'Jan 01 1999')

SET @store_nbr =
CASE WHEN @counter < 999 THEN '6380'
     WHEN @counter BETWEEN 1000 AND 2999 THEN '7066'
     WHEN @counter BETWEEN 3000 AND 3999 THEN '7067'
     WHEN @counter BETWEEN 4000 AND 6999 THEN '7131'
     WHEN @counter BETWEEN 7000 AND 7999 THEN '7896'
     WHEN @counter BETWEEN 8000 AND 9999 THEN '8042'
     ELSE '6380'
END

SET @qty_sold =
CASE WHEN @counter BETWEEN 0 AND 2999 THEN 11
     WHEN @counter BETWEEN 3000 AND 5999 THEN 23
     ELSE 37
END

SET @terms =
CASE WHEN @counter BETWEEN 0 AND 2999 THEN 'Net 30'
WHEN @counter BETWEEN 3000 AND 5999 THEN 'Net 60'
ELSE 'On Invoice'
END

-- SET @title = (SELECT title_id FROM big_sales WHERE qty = (SELECT MAX(qty)
FROM big_sales))

SET @title =
CASE WHEN @counter < 999 THEN 'MC2222'
     WHEN @counter BETWEEN 1000 AND 1999 THEN 'MC2222'
     WHEN @counter BETWEEN 2000 AND 3999 THEN 'MC3026'
     WHEN @counter BETWEEN 4000 AND 5999 THEN 'PS2106'
     WHEN @counter BETWEEN 6000 AND 6999 THEN 'PS7777'
     WHEN @counter BETWEEN 7000 AND 7999 THEN 'TC3218'
     ELSE 'PS1372'
END

-- PRINT @separator
-- SELECT @message = STR( @counter, 10 ) -- + STR( SQRT( CONVERT( FLOAT,
          @counter ) ), 10, 4 )

-- PRINT @message

BEGIN TRAN
INSERT INTO [pubs].[dbo].[big_sales]([stor_id], [ord_num], [ord_date],
[qty], [payterms], [title_id])
VALUES(@store_nbr, CAST(@ord_nbr AS CHAR(5)), @order_date, @qty_sold,
@terms, @title)
COMMIT TRAN
END

SET @message = CAST(DATEDIFF(ms, @starttime, GETDATE()) AS VARCHAR(20))
PRINT @message

/*
TRUNCATE table big_sales
INSERT INTO big_sales
SELECT * FROM sales
SELECT title_id, sum(qty)
FROM big_sales
```

```
group by title_id
order by sum(qty)

SELECT * FROM big_sales
*/
```

При использовании *SET NOCOUNT OFF*, время выполнения равнялось 5176 миллисекундам. А при использовании *SET NOCOUNT ON* время выполнения – 1620 миллисекунд!

Подумайте о том, чтобы в начале каждой хранимой процедуры или скрипта, которые не требуют подсчета числа обработанных строк для выходного потока, использовать *SET NOCOUNT ON*.

TOP и SET ROWCOUNT

Предложение TOP оператора *SELECT* накладывает ограничение на количество строк, возвращаемых отдельным запросом, в то время как *SET ROWCOUNT* накладывает ограничение на количество строк, обрабатываемых всеми последующими запросами. Эти команды обеспечивают хорошую эффективность для многочисленных задач программирования.

SET ROWCOUNT устанавливает максимальное число строк, которые могут быть обработаны операторами *SELECT*, *INSERT*, *UPDATE* и *DELETE*. Установка вступает в действие сразу после выполнения команды и распространяется только на текущий сеанс. Для удаления данного ограничения используется *SET ROWCOUNT 0*.

В некоторых практических задачах более эффективным является использование *TOP* или *SET ROWCOUNT*, чем стандартных команд SQL. Покажем это на нескольких примерах.

TOP n

Одним из самых популярных запросов практически в любой базе данных является запрос на получение первых *n* элементов из списка. В случае с базой данных *pubs* можно было бы найти первые 5 наиболее популярных названий. Сравните три решения этой задачи – с использованием *TOP*, *SET ROWCOUNT* и *ANSI SQL*.

«Чистый» ANSI SQL

```
SELECT title, ytd_sales
FROM titles a
WHERE ( SELECT COUNT(*)
        FROM titles b
        WHERE b.ytd_sales > a.ytd_sales
      ) < 5
ORDER BY ytd_sales DESC
```

Решение с использованием «чистого» ANSI SQL выполняет коррелированный подзапрос, что может оказаться неэффективным, особенно в том случае, если на таблице ytd_sales не используется никаких индексов, которые бы могли его улучшить. Кроме того, команда «чистого» ANSI SQL не убирает NULL-значения в таблице ytd_sales , и при этом не делает различия в случае, когда имеется много совпадающих значений.

Использование SET ROWCOUNT:

```
SET ROWCOUNT 5

SELECT title, ytd_sales
  FROM titles
  ORDER BY ytd_sales DESC

SET ROWCOUNT 0
```

Использование TOP n:

```
SELECT TOP 5 title, ytd_sales
  FROM titles
  ORDER BY ytd_sales DESC
```

Второе решение, использующее *SET ROWCOUNT*, прерывает выполнение запроса *SELECT*, в то время как третий вариант решения, использующий *TOP n*, завершает свою работу после того, как найдены первые 5 строк. В этом случае также используется предложение *ORDER BY*, выполняющее сортировку всей таблицы, прежде чем будут найдены конечные результаты. Оба запроса фактически имеют одинаковые планы выполнения запроса. Однако основное преимущество *TOP* перед *SET ROWCOUNT* заключается в том, что *SET* должен обработать рабочую таблицу, как этого требует предложение *ORDER BY*, в то время как для *TOP* это не требуется.

На большой таблице, чтобы избежать сортировки, можно было бы создать индекс для ytd_sales. Запрос тогда использовал бы индекс для нахождения первых пяти строк и далее бы не выполнялся. Сравним это решение с первым, которое просматривало всю таблицу и выполняло коррелированный подзапрос для каждой строки. Для маленькой таблицы различия в эффективности работы незначительны, но для большой продолжительность обработки первого запроса может составить часы, а последних двух – секунды.

При определении того, что необходимо в запросе, проверьте, не нужны ли вам всего несколько строк из результата. И если да, то предложение TOP окажется верным средством для экономии времени сервера.

Использование инструментальных средств для настройки операторов SQL

В предыдущих разделах описывалось, как можно применять различные технические приемы Microsoft SQL Server для настройки операторов SQL. Использование инструментальных средств при настраивании операторов SQL является решающим моментом в повышении производительности и устранении ошибок пользователя. В этом разделе будет показано, как можно использовать различные инструментальные средства для повышения скорости выполнения и улучшения производительности.

Microsoft Query Analyzer

Microsoft SQL Server включает в себя сервисную программу Query Analyzer, которая дает возможность пользователям писать и выполнять операторы SQL и скрипты T-SQL. Query Analyzer графически отображает планы выполнения перед или после выполнения операторов SQL. Опция Display Estimated Execution Plan (отображение приблизительного плана выполнения) в меню Query (запрос) отображает план выполнения запроса, который SQL Server будет использовать для выполнения оператора SQL. Опция Show Execution Plan (отображение плана выполнения) в меню Query показывает план запроса, который SQL Server использовал при выполнении запроса SQL. Графический план выполнения использует иконки для обозначения шагов и методов извлечения данных, которые SQL Server выбирает для выполнения оператора SQL. План выполнения – это графическое представление табличных данных, выводимых при помощи операторов SET SHOWPLAN_ALL и SET SOWPLAN_TEXT (Рисунок 1).

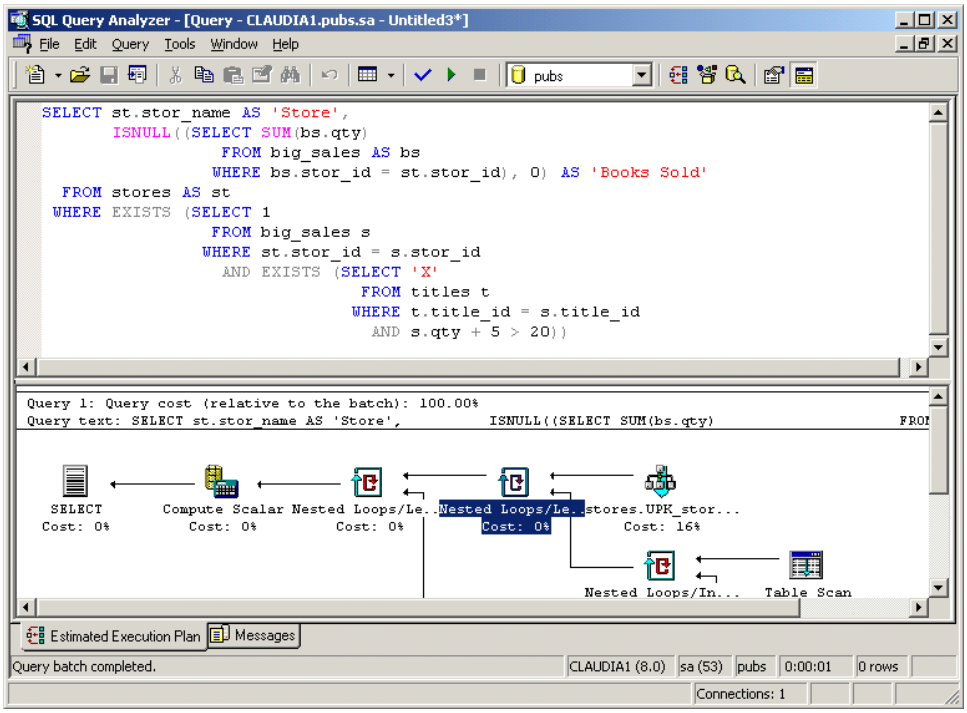

Рисунок 1. Графический план выполнения в Query Analyzer

При изучении операций на плане выполнения можно понять характеристики производительности операторов SQL и обнаружить необходимость оптимизации.

Планы выполнения могут значительно усложняться при работе со сложными операторами SQL. В этом случае пользователю сложнее читать план выполнения и обнаруживать неэффективно работающие места.

Если вы решили, что оператор SQL нуждается в оптимизации, то можете воспользоваться Query Analyzer, чтобы вручную настроить оператор SQL. Для того чтобы вручную настроить оператор SQL, вам необходимо открыть новое окно в Query Analyzer, переработать оператор SQL, используя некоторые из технических приемов, представленных в данном документе, пересмотреть план выполнения и выполнить оператор SQL для того, чтобы получить время выполнения. Затем вручную можно этот процесс повторять до тех пор, пока не будет найден альтернативный оператор SQL с удовлетворительной производительностью. Ограниченность такого подхода заключается в том, что при переформулировании сложного оператора SQL необходимо анализировать эффективность работы SQL, а при изменении кода SQL эксперт может допустить ошибку, которая в свою очередь может привести к альтернативному оператору SQL, который не возвращает тот же результирующий набор, что и первоначальный оператор SQL.

Другой ограниченностью является длительность процесса, выполняемого вручную. Количество альтернативных операторов SQL, которые пользователь может обработать, ограничено временем, которое пользователь может потратить на выяснение новых способов написания оператора SQL. Чтобы сделать процесс настройки SQL более эффективным, избегажать ошибок пользователя и сэкономить время, желательно использовать инструментарий, который автоматизирует процесс настройки запросов.

Quest Central для SQL Server SQL Tuning

Quest Central® для SQL Server – это интегрированное решение для управления базами данных, которое упрощает ежедневные задачи и включает в себя набор инструментов, позволяющий пользователям достичь более высокого уровня в обеспечении доступности и надежности. Quest Central для SQL Server включает в себя Database Administration (администрирование баз данных), Space Management (управление дисковым пространством), Database Analysis (анализ баз данных) и SQL Tuning (настройка SQL).

SQL Server SQL Tuning в Quest Central включает в себя отображение графического плана выполнения, SQL Scanner (сканер), который в активном режиме обнаруживает проблематичные операторы SQL непосредственно в самих объектах базы данных или в исходном коде, и SQL Optimizer, который автоматически переписывает операторы SQL в каждом из возможных вариантов, позволяя определить наиболее эффективный вариант SQL для конкретной среды окружения базы данных.

Обычно приложения баз данных содержат тысячи операторов SQL. Операторы SQL могут быть расположены в таких объектах базы данных, как представления и хранимые процедуры, или в исходном коде приложения. Без автоматизированного инструмента процесс извлечения и просмотра каждого оператора SQL вручную весьма утомителен и трудоемок. Модуль SQL Scaner автоматизирует процесс извлечения и просмотра операторов SQL непосредственно из исходного кода и предлагает проактивный подход обнаружения потенциальных проблемных производительности SQL запросов (Рисунок 2).

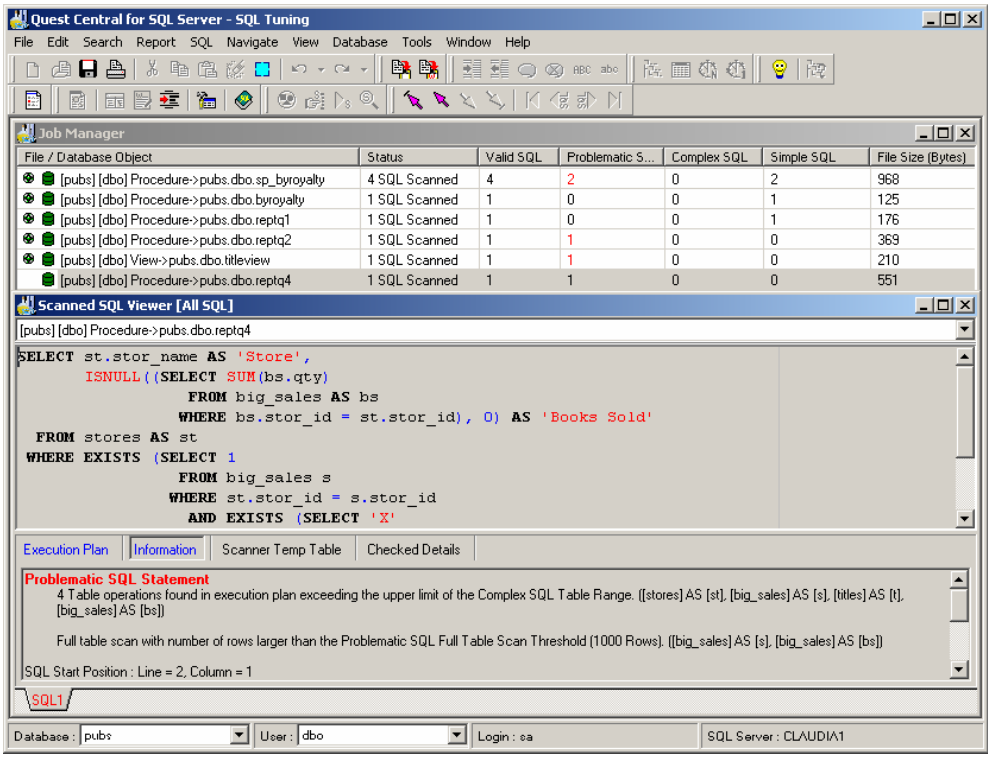

Рисунок 2. SQL Scaner, анализирующий множество операторов SQL для определения проблем производительности

SQL Scaner извлекает операторы SQL, встроенные в объекты базы данных, исходный код и файлы/таблицы трасы Microsoft SQL Server Profiler фиксирует файлы/таблицы, не выполняя никаких программ. SQL Scaner может извлекать операторы SELECT, SELECT..INTO, INSERT, DELETE и UPDATE. SQL Scaner анализирует в пакетном режиме планы выполнения каждого оператора SQL и группирует их в зависимости от уровня сложности и от уровня предполагаемых проблем производительности. Планы выполнения, содержащие неэффективные операции и операции, которые могут вызвать большое число операций ввода-вывода, такие как полное сканирование больших таблиц, полное сканирование таблиц во вложенных циклах или сканирование множества таблиц, классифицируется как Проблемные. В этом случае SQL Scaner позволяет предварительно определить проблемные с точки зрения места в коде SQL.

Т.к. язык SQL достаточно сложный, то существует множество способов написания оператора SQL, которые возвращают один и тот же результирующий набор. Однако небольшие различия в коде SQL могут значительно повлиять на его производительность. SQL Optimizer использует механизм преобразования SQL, который полностью преобразует оператор SQL во все возможные его эквивалентные разновидности, сохраняя при этом ту же логику. Процесс перезаписи SQL включает использование синтаксических SQL-преобразований и подсказок SQL Server, которые являются дополнительной возможностью для пользователя.

После преобразования SQL-оператора SQL Optimizer получает план выполнения для каждого оператора SQL и приводит оптимизированные операторы к операторам с уникальным планом выполнения, т.к. план выполнения – это именно то, что определяет производительность оператора SQL. Этот масштабный процесс преобразования SQL проходит на персональном компьютере и не затрагивает ресурсы сервера баз данных (рисунок 3).

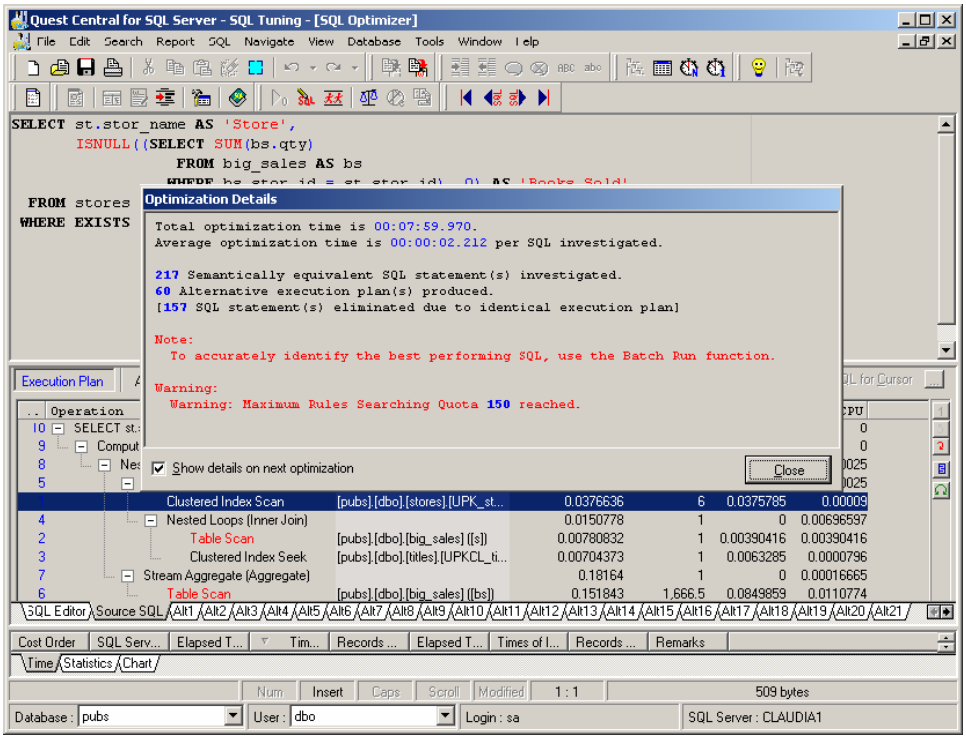

Рисунок 3. SQL Optimizer автоматически переписывает операторы SQL

После завершения процесса оптимизации, SQL Optimizer выводит список альтернативных вариантов запросов SQL, планы выполнения и стоимость каждого плана выполнения для SQL Server. Пользователь может просмотреть эти альтернативы и решить, какие из них использовать в базе данных для получения наилучших показателей по времени исполнения и количеству операций ввода-вывода для конкретного окружения базы данных. После того, как был определен наиболее эффективный оператор SQL, пользователи могут активизировать SQL Comparer для синхронного сравнения альтернатив, чтобы отобразить синтаксис, план выполнения и статистику времени выполнения.

ЗАКЛЮЧЕНИЕ

Производительность приложения в среде Microsoft SQL Server непосредственно связана с эффективностью используемых в ней операторов SQL. В этой статье было описано несколько технических приемов Microsoft SQL Server, применяемых для настройки операторов SQL. Оптимизация операторов SQL вручную или с использованием встроенных утилит SQL Server – трудоемкая и требующая глубоких знаний задача. Quest Central for SQL Server–SQL Tuning предлагает решения, которые автоматизируют процесс настройки SQL, экономят время администратора баз данных и разработчика, не требуют большого опыта и знаний, увеличивая производительность их работы и повышая эффективность выполнения операторов SQL, во всех ваших системах Microsoft SQL Server.