O'REILLY®

SQL Server

Наладка и оптимизация для профессионалов

Дмитрий Короткевич

SQL Server Advanced Troubleshooting and Performance Tuning

Best Practices and Techniques

Dmitri Korotkevich

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

SQL Server

Наладка и оптимизация
для профессионалов

Дмитрий Короткевич

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.988.02-018.2
УДК 004.383.2
К68

Короткевич Дмитрий

К68 SQL Server. Наладка и оптимизация для профессионалов. — СПб.: Питер, 2023. — 512 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2332-2

Исчерпывающий обзор лучших практик по устранению неисправностей и оптимизации производительности Microsoft SQL Server. Специалисты по базам данных, в том числе разработчики и администраторы, научатся выявлять проблемы с производительностью, системно устранять неполадки и расставлять приоритеты при тонкой настройке, чтобы достичь максимальной эффективности.

Автор книги Дмитрий Короткевич — Microsoft Data Platform MVP и Microsoft Certified Master (MCM) — расскажет о взаимозависимостях между компонентами баз данных SQL Server. Вы узнаете, как быстро провести диагностику системы и найти причину любой проблемы. Методы, описанные в книге, совместимы со всеми версиями SQL Server и подходят как для локальных, так и для облачных конфигураций SQL Server.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018.2
УДК 004.383.2

ISBN 978-1098101862 англ.

Authorized Russian translation of the English edition of SQL Server Advanced Troubleshooting and Performance Tuning, ISBN 9781098101923

ISBN 978-5-4461-2332-2

Краткое содержание

Оглавление

Предисловие

Прошло уже несколько лет с тех пор, как была издана моя предыдущая книга. За это время многое изменилось. Выпущено несколько новых версий SQL Server. Продукт стал более зрелым, кросс-платформенным, в нем появилась полноценная поддержка облачных технологий. Но я все равно не торопился публиковать новое издание книги «Pro SQL Server Internals» («Внутреннее устройство SQL Server для профессионалов»).

Тому было несколько причин. Как бы ни были хороши новые функции, они не меняли *фундаментальных* принципов работы продукта. Материал из моих старых книг по большей части подходит к SQL Server 2017, SQL Server 2019 и даже к свежему SQL Server 2022. Что еще более важно, я хотел написать книгу по-другому.

Наверное, стоит уточнить. Как некоторые из вас, возможно, знают, я уже много лет провожу курсы по SQL Server и использую собственные книги как методические пособия к этим курсам. Собственно, я и писать начал именно потому, что хотел представить материал в более структурированном формате, чем презентации PowerPoint. Я рад, что моим читателям это понравилось и книги оказались полезными.

Все мои курсы были посвящены внутреннему устройству SQL Server. Я всегда считал, что профессионал должен разбираться в своих инструментах, чтобы достичь успеха. Я рассказываю ученикам, как работает SQL Server, и помогаю им применять эти знания и создавать эффективные системы. Но в какой-то момент я обнаружил, что самой популярной темой на моих занятиях стали устранение неполадок и настройка производительности: ученикам интересно, когда я описываю конкретную *проблему*, а затем объясняю, *почему* она возникла.

Изменив методику преподавания, я решил изменить и методику написания книги. Прошло 18 месяцев — и результат перед вами. Лично мне нравится то, что получилось. Книга по-прежнему посвящена внутреннему устройству SQL Server, но теперь она лаконичнее и практичнее моих предыдущих работ[1]. Она научит вас

[1] Название книги также изменилось. Теперь она называется «SQL Server Advanced Troubleshooting and Performance Tuning» («SQL Server. Наладка и оптимизация для профессионалов»). — *Примеч. ред.*

эффективно обнаруживать и устранять типичные проблемы при работе с SQL Server, не перегружая лишней информацией. Книга также покажет, в каком направлении двигаться, если вы хотите узнать еще больше.

Здесь описана методология, которую используют многие консультанты мирового уровня по SQL Server. Вы научитесь собирать и анализировать данные, выявлять узкие места и очаги неэффективности. Что еще более важно, я покажу, как рассматривать систему *целостно* и «видеть лес за деревьями».

Содержание книги не привязано к какой-либо конкретной версии SQL Server. За немногими исключениями оно актуально для всех версий от SQL Server 2005 до SQL Server 2022 и последующих версий. Материал также подходит для управляемых служб SQL Server, работающих в облаке.

Для кого эта книга

Когда меня спрашивают, для кого предназначены мои книги, я всегда говорю, что пишу для *специалистов по базам данных*. Я намеренно использую такой термин, потому что считаю, что границы, разделяющие администраторов баз данных, разработчиков баз данных и даже разработчиков приложений, довольно условны. Сегодня невозможно добиться успеха в IT, если ограничиваться узкой специализацией и не расширять сферу своей компетенции и ответственности.

Владеть широким спектром технологий особенно важно в культуре DevOps, где команды разрабатывают и поддерживают решения сами для себя. Для разработчиков становится обычным делом устранять проблемы с производительностью, которые могут быть связаны с инфраструктурой или неэффективным кодом базы данных.

В общем, в какой бы роли вы ни работали с SQL Server, эта книга для вас. Я надеюсь, что вы найдете для себя полезную информацию независимо от того, как называется ваша должность.

Еще раз спасибо за ваше доверие, и я надеюсь, что вы прочтете эту книгу с таким же удовольствием, с каким я ее писал!

Структура книги

Книга состоит из 16 глав:

Глава 1 «Установка и настройка SQL Server» содержит принципы и лучшие методики того, как выбирать оборудование и настраивать экземпляры SQL Server.

Глава 2 «Модель выполнения SQL Server и статистика ожидания» описывает SQLOS — очень важный компонент SQL Server. Здесь же вы познакомитесь с таким распространенным методом устранения неполадок, как статистика ожидания. На эту главу опирается весь остальной материал книги.

Глава 3 «Производительность дисковой подсистемы» дает представление о том, как SQL Server взаимодействует с подсистемой ввода/вывода и как анализировать и оптимизировать ее производительность.

Глава 4 «Неэффективные запросы» демонстрирует несколько методов того, как выявлять неэффективные запросы и выбирать целевые объекты для оптимизации запросов.

Глава 5 «Хранение данных и настройка запросов» объясняет, как SQL Server работает с данными в базе данных, и дает рекомендации по настройке запросов.

Глава 6 «Загрузка процессора» рассматривает распространенные причины высокой загрузки ЦП и учит бороться с узкими местами на уровне процессора.

Глава 7 «Проблемы с оперативной памятью» посвящена настройкам SQL Server, относящимся к памяти, и описывает, как анализировать использование памяти и решать связанные с ней проблемы.

Глава 8 «Блокировки и конкурентный доступ» рассказывает о модели конкурентного доступа, используемой в SQL Server, и о том, как обращаться с блокировками в системе.

Глава 9 «Работа с базой данных tempdb и ее производительность» описывает использование системной базы данных `tempdb` и лучшие методики ее конфигурации. Кроме того, здесь содержатся рекомендации о том, как оптимально использовать вре́менные объекты и устранять распространенные узкие места в `tempdb`.

Глава 10 «Кратковременные блокировки» посвящена кратковременным блокировкам в SQL Server. Рассматриваются случаи, когда они вызывают проблемы, и способы решения этих проблем.

Глава 11 «Журнал транзакций» рассказывает о том, как устроен журнал транзакций в SQL Server и как избавиться от распространенных узких мест и ошибок в нем.

Глава 12 «Группы доступности AlwaysOn» рассматривает самую популярную технологию высокой доступности SQL Server и частые проблемы, с которыми можно столкнуться при ее использовании.

Глава 13 «Другие примечательные типы ожиданий» описывает несколько распространенных типов ожиданий, которые не рассматривались в прочих главах.

Глава 14 «Анализ схемы базы данных и индексов» дает ряд советов о том, как обнаруживать неэффективные участки структуры базы данных, а также оценивать использование индексов и их работоспособность.

Глава 15 «SQL Server в виртуализированных средах» рассказывает о передовых методах настройки виртуальных экземпляров SQL Server и устранении сопутствующих неполадок.

Глава 16 «SQL Server в облаке» описывает, как настраивать и использовать SQL Server в облачных виртуальных машинах. В ней также представлен обзор управляемых служб SQL Server, доступных в Microsoft Azure, Amazon Web Services (AWS) и Google Cloud Platform (GCP).

В конце каждой главы приведен контрольный список наиболее важных шагов по устранению неполадок, связанных с темой главы.

Наконец, **приложение «Типы ожиданий»** можно использовать как справочник по распространенным типам ожиданий и методам устранения основных неполадок для каждого типа.

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины или важные понятия.

`Моноширинный шрифт`

Используется для листингов программного кода, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова.

`Полужирный моноширинный`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Моноширинный курсив`

Показывает текст, который следует заменить значениями, полученными от пользователя или из контекста.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок и других элементов интерфейса, каталогов, имен и расширений файлов.

Совет или предложение.

Общее примечание.

Предупреждение или предостережение.

Использование исходного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу https://github.com/aboutsqlserver/code.

В папке `Troubleshooting Scripts` вы найдете записные книжки Azure Data Studio[1] с использованными в книге сценариями диагностики и устранения неполадок. Примеры сценариев и приложений также есть в папке `Companion Materials (Books)`.

Если не указано иное, сценарии будут работать во всех версиях SQL Server, начиная с SQL Server 2005. В старых версиях могут не поддерживаться некоторые столбцы динамических административных представлений, и вам придется закомментировать их.

Я планирую поддерживать и расширять библиотеку диагностических сценариев, так что проверяйте обновления в репозитории.

Если у вас возникнут вопросы технического характера по использованию примеров кода, направляйте их по электронной почте на адрес bookquestions@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя

[1] Azure Data Studio можно скачать с сайта Microsoft: https://oreil.ly/zwwCf.

данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Как связаться с автором

Вы можете написать мне по адресу dk@aboutsqlserver.com, если у вас есть вопросы по книге или по SQL Server в целом. Я всегда рад помочь, чем смогу.

Также загляните в мой блог по адресу https://aboutsqlserver.com. Обещаю писать туда чаще, раз уж книга наконец вышла!

Благодарности

Прежде всего, как и всегда, я хотел бы поблагодарить свою семью за постоянную помощь и поддержку. Писательство — это идеальное оправдание, чтобы избежать домашних обязанностей. Я до сих пор не понимаю, почему мне все сошло с рук!

Кроме того, я чрезвычайно благодарен Эрланду Соммарскогу (*Erland Sommarskog*), Томасу Грозеру (*Thomas Grohser*) и Уве Рикену (*Uwe Ricken*), которые проделали большую работу, рецензируя эту книгу. Благодаря им она стала значительно лучше и обрела окончательную форму.

Эрланд Соммарског работает с SQL Server уже тридцать лет и имеет статус Microsoft Data Platform MVP с 2001 года. Он работает независимым консультантом в Стокгольме (Швеция). Эрланд с удовольствием делится знаниями и опытом с сообществом. В свободное от SQL Server время он играет в бридж и путешествует.

Томас Грозер работает IT-специалистом более 35 лет и уже 12 лет является Microsoft Data Platform MVP. Он использует SQL Server с 1994 года и специализируется на архитектуре и реализации высокозащищенных, доступных, восстанавливаемых и эффективных баз данных, а также их базовой инфраструктуры. В свободное время Томас любит делиться знаниями, накопленными за десятилетия, с сообществом SQL Server и платформ данных, выступая в группах пользователей и на конференциях по всему миру.

Уве Рикен — Microsoft Data Platform MVP и Microsoft Certified Master (SQL Server 2008) из Франкфурта (Германия). Уве работает с SQL Server с 2007 года

и специализируется на внутреннем устройстве баз данных и индексировании, а также на архитектуре и разработке баз данных. Он регулярно выступает на конференциях и мероприятиях по SQL Server и ведет блог http://www.sqlmaster.de.

Спасибо вам, Эрланд, Томас и Уве! Работать с вами было очень круто!

Огромное спасибо моему коллеге Андре Фиано (*Andre Fiano*) — одному из самых знающих специалистов по инфраструктуре, которых я когда-либо встречал. Я многому научился у Андре, и он помог мне подготовить несколько наглядных примеров для этой книги.

И конечно же, я хотел бы поблагодарить всю команду O'Reilly и особенно Сару Грей (*Sarah Grey*), Элизабет Келли (*Elizabeth Kelly*), Кейт Дулли (*Kate Dullea*), Кристен Браун (*Kristen Brown*) и Одри Дойл (*Audrey Doyle*). Спасибо вам за то, что помогли привести мой английский в приемлемую форму и убедили меня, будто я умею рисовать диаграммы!

Эта книга посвящена SQL Server, и я хочу поблагодарить команду Microsoft за усердную работу над этим продуктом. Мне очень интересно, как он будет развиваться дальше.

И последнее, но не менее важное: отдельная благодарность — всем моим друзьям из сообщества #SQLFamily, которые поддерживали и подбадривали меня! Писать для такой замечательной аудитории — сплошное удовольствие!

Спасибо вам всем!

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Установка и настройка SQL Server

Серверы баз данных функционируют не в безвоздушном пространстве. Они входят в экосистему одного или нескольких клиентских приложений. Базы данных приложений размещаются на одном или нескольких экземплярах SQL Server, а они, в свою очередь, развернуты на физическом или виртуальном оборудовании. Данные хранятся на дисках, чей ресурс обычно приходится делить с другими системами баз данных и не только. Наконец, всем компонентам нужна сеть, чтобы обмениваться данными и сохранять их.

Бороться с неполадками в базах данных непросто из-за сложности их экосистем и внутренних зависимостей. С точки зрения клиентов, большинство проблем касаются производительности: приложения работают медленно и не отвечают на запросы, время ожидания истекает, а иногда приложения вообще не подключаются к базе данных. Настоящая причина проблемы может скрываться где угодно. Может быть, аппаратное обеспечение неисправно или неправильно настроено, а может, в базе данных неэффективные индексы, схемы или код. Возможно, SQL Server перегружен, а может, клиентское приложение работает с ошибками или плохо спроектировано. Все это означает, что для поиска и устранения проблем нужно целостное представление обо всей системе.

Эта книга посвящена устранению неполадок в SQL Server. Но устранение неполадок всегда стоит начинать с анализа экосистемы приложения и среды SQL Server. В этой главе даны рекомендации о том, как выполнять этот анализ и обнаруживать наиболее распространенные проблемы в конфигурации SQL Server.

Сперва я расскажу о настройке оборудования и операционной системы. Затем пойдет речь об установке SQL Server и конфигурации базы данных. Далее мы коснемся темы консолидации SQL Server и дополнительных издержек, возникающих из-за средств мониторинга.

Аппаратное обеспечение и операционная система

Обычно устранять неполадки и настраивать производительность приходится в действующих системах, которые хранят много данных и работают под высокой

нагрузкой. Тем не менее невозможно вовсе не говорить об аппаратной части. К тому же в ходе устранения неполадок может обнаружиться, что серверы просто не справляются с нагрузкой и нуждаются в обновлении.

Я не буду рекомендовать конкретных производителей или модели комплектующих, потому что аппаратное обеспечение быстро совершенствуется и к моменту публикации книги любой совет попросту устареет. Вместо этого я выскажу универсальные соображения, которые, надеюсь, еще долго будут актуальными.

Центральный процессор

Самая затратная часть системы — это, безусловно, лицензия на коммерческое ядро базы данных. Как правило, она значительно дороже оборудования, на котором предполагается разворачивать сервер БД. Поэтому рекомендую покупать самый мощный ЦП, какой позволит ваш бюджет, особенно если вы используете не версию SQL Server Enterprise Edition (эта версия не ограничивает количество доступных ядер).

Обратите внимание на модель процессора. Каждое новое поколение процессоров производительнее предыдущего. Можно получить прирост производительности на 10–15 %, просто поставив новый ЦП, даже если у него такая же тактовая частота, как у старого.

Иногда, когда стоимость лицензии — не главная проблема, приходится выбирать, что лучше: более медленный процессор с бóльшим количеством ядер или более быстрый процессор с меньшим количеством ядер. В этом случае решение во многом зависит от загруженности системы. Для систем оперативной обработки транзакций (OLTP), особенно In-Memory OLTP, выгоднее будет одноядерный высокопроизводительный процессор. С другой стороны, для хранилищ данных и аналитических задач больше подойдет высокая степень параллелизма и большое количество ядер.

Оперативная память

В сообществе SQL Server бытует такая шутка:

— *Сколько памяти нужно для SQL Server?*

— *Больше*.

В этой шутке есть доля правды. Большой объем памяти позволяет SQL Server кэшировать больше данных. Это, в свою очередь, сокращает количество дисковых операций ввода/вывода (I/O) и положительно сказывается на производительности. Поэтому увеличение объема памяти сервера — зачастую самый дешевый и быстрый способ решить некоторые проблемы с производительностью.

Например, предположим, что система страдает от неоптимизированных запросов. Казалось бы, их влияние можно уменьшить, если добавить памяти и таким образом уменьшить чтение с физического диска для этих запросов. Но очевидно, что это не решает основную проблему и к тому же опасно, потому что данные могут разрастись до того, что перестанут помещаться в кэш. Тем не менее в качестве временного решения такой подход иногда годится.

У SQL Server Enterprise Edition объем используемой памяти не ограничен. У других версий есть ограничения. Standard Edition (SQL Server 2014 и более поздних версий) может использовать до 128 Гбайт ОЗУ для буферного пула, 32 Гбайт ОЗУ на каждую базу данных In-Memory OLTP и 32 Гбайт ОЗУ для хранения сегментов индекса columnstore. В Web Edition доступно вдвое меньше памяти, чем в Standard Edition. Учитывайте эти ограничения, когда собираете или обновляете экземпляры SQL Server, отличные от Enterprise Edition. Не забудьте выделить дополнительную память для других компонентов SQL Server, например кэша планов и менеджера блокировок.

Короче, добавьте столько памяти, сколько можете себе позволить. В наше время это дешево. Если ваши базы данных небольшие, то чрезмерное количество памяти ни к чему, однако учитывайте, что в будущем объем данных может вырасти.

Дисковая подсистема

Для хорошей производительности SQL Server необходима исправная и быстрая дисковая подсистема. SQL Server очень интенсивно занимается вводом/выводом, то есть постоянно считывает и записывает данные на диск.

Архитектуру дисковой подсистемы для SQL Server можно построить по-разному. Главное — добиться, чтобы задержка запросов ввода/вывода была минимальной. Для критически важных систем первого класса надежности я рекомендую, чтобы задержка чтения и записи данных не превышала 3–5 мс, а для записи журнала транзакций — 1–2 мс. К счастью, этих показателей легко достичь с помощью флеш-накопителей.

Но есть загвоздка: анализируя производительность ввода/вывода в SQL Server, нужно измерять время задержки на уровне самого́ SQL Server, а не на уровне хранилища. В SQL Server задержки могут оказаться значительно дольше, чем ключевые метрики производительности хранилища (KPI), потому что при интенсивном вводе/выводе могут возникать очереди. (В главе 3 мы рассмотрим, как собирать и анализировать данные о производительности ввода/вывода.)

Если ваша подсистема хранения поддерживает несколько уровней производительности, я рекомендую разместить на самом быстром диске базу данных `tempdb`, а на оставшихся — журнал транзакций и файлы данных. База данных `tempdb` — это общий ресурс на сервере, и для нее важна хорошая пропускная способность ввода/вывода.

Записи в файлы журнала транзакций выполняются синхронно. Для этих файлов важна низкая задержка записи. Записи в журнал транзакций также производятся последовательно; однако помните, что размещение нескольких файлов журналов и/или файлов данных на одном диске чревато режимом произвольного доступа сразу в нескольких базах данных.

Я рекомендую помещать файлы данных и журналов на разные физические диски, потому что при этом базу удобнее обслуживать и легче восстанавливать. Но стоит учитывать физическую конфигурацию хранилища. В некоторых случаях, когда в дисковых массивах недостаточно шпинделей, разделение массива на несколько LUN может снизить производительность всего массива.

В своих системах я не разбиваю кластеризованные и некластеризованные индексы по нескольким файловым группам, размещая их на разных дисках. От этого редко увеличивается производительность ввода/вывода, если только вы не разделяете полностью пути хранения по файловым группам. В то же время такая конфигурация может значительно усложнить аварийное восстановление.

Наконец, помните, что для некоторых технологий SQL Server важна хорошая эффективность последовательного ввода/вывода. Например, в In-Memory OLTP вообще не используется произвольный доступ, и ограничивающим фактором при запуске и восстановлении базы данных становится производительность последовательного чтения. Обход хранилища данных тоже зависит от последовательного ввода/вывода, когда B-деревья и индексы columnstore не сильно фрагментированы. У флеш-памяти разница между производительностью последовательного и произвольного ввода/вывода незначительна, а вот у магнитных дисков она довольно велика.

Сеть

SQL Server связывается с клиентами и другими серверами по сети. Очевидно, нужна достаточная пропускная способность сети, чтобы поддерживать эту связь. Остановлюсь на нескольких важных деталях.

Во-первых, при устранении неполадок, связанных с производительностью сети, необходимо анализировать топологию всей сети. Помните, что пропускная способность сети ограничена скоростью ее самого медленного компонента. Например, у вас может быть 10-гигабитный восходящий канал от сервера, но если где-то в сети оказался коммутатор на 1 Гбит/с, он ограничит общую пропускную способность. Это особенно важно для сетевых хранилищ: убедитесь, что пути доступа к дискам максимально эффективны.

Во-вторых, сложилась общепринятая практика выделять отдельную сеть для передачи тактового импульса в отказоустойчивых кластерах AlwaysOn и группах доступности AlwaysOn. Иногда стоит подумать о выделении отдельной сети для всего трафика группы доступности. Этот подход повышает надежность

кластеров в простых конфигурациях, когда все кластерные узлы принадлежат одной подсети и могут использовать маршрутизацию уровня 2. Но в сложных конфигурациях с множеством подсетей наличие нескольких сетей может вызвать проблемы маршрутизации. Работая с такими конфигурациями, будьте осторожны и проверяйте, что связь между узлами сети налажена правильно, особенно в виртуальных средах, о которых я расскажу в главе 15.

Виртуализация добавляет еще один уровень сложности. Рассмотрим ситуацию, когда у вас есть виртуальный кластер SQL Server, узлы которого работают на разных хостах. Вам нужно будет убедиться, что хосты могут разделять и маршрутизировать трафик в кластерной сети отдельно от клиентского трафика. Если весь трафик локальной сети обслуживается через одну физическую сетевую карту, то тактовые импульсы теряют смысл.

Операционные системы и приложения

Как правило, я рекомендую использовать самую свежую версию операционной системы, которая поддерживает вашу версию SQL Server. Убедитесь, что и ОС, и SQL Server обновлены до последних версий, и наладьте регулярную установку обновлений.

Если вы используете старую версию SQL Server (до 2016), лучше устанавливать 64-разрядную ОС. В большинстве случаев 64-разрядная версия работает эффективнее 32-разрядной и лучше переносит масштабирование оборудования.

Начиная с SQL Server 2017, сервер баз данных можно развертывать и на Linux. С точки зрения производительности версии SQL Server для Windows и Linux очень похожи. Выбор ОС зависит от корпоративной экосистемы и от того, какую систему вам удобнее поддерживать. Имейте в виду, что для развертывания на Linux может потребоваться несколько иная стратегия высокой доступности (HA, High Availability) по сравнению с Windows. Например, для автоматического аварийного переключения, возможно, придется применять Pacemaker вместо Windows Server Failover Cluster (WSFC).

По возможности лучше использовать выделенный хост SQL Server. Помните, что проще и дешевле масштабировать серверы приложений и не тратить ценные ресурсы на хост базы данных.

В то же время не следует запускать на сервере несущественные процессы. Например, многие специалисты по базам данных запускают SQL Server Management Studio (SSMS) только на удаленных рабочих столах. Всегда лучше работать удаленно и не потреблять ресурсы сервера.

Наконец, если на сервере должно работать антивирусное ПО, то все папки баз данных нужно исключить из сканирования.

Виртуализация и облачные технологии

Современная IT-инфраструктура опирается на виртуализацию, которая обеспечивает дополнительную гибкость, упрощает управление и снижает затраты на оборудование. Поэтому чаще всего вам придется работать с виртуализированной инфраструктурой SQL Server.

Ничего плохого в этом нет. Грамотно реализованная виртуализация дает множество преимуществ при приемлемом снижении производительности. В случае VMware vSphere vMotion или Hyper-V Live Migration виртуализация добавляет еще один уровень высокой доступности. Виртуализация позволяет плавно обновлять аппаратное обеспечение и упрощает управление базой данных. Если вам не требуется выжимать максимум из оборудования, то экосистему SQL Server лучше виртуализировать.

На больших серверах с большим количеством ЦП накладные расходы на виртуализацию увеличиваются. Однако во многих случаях это оказывается вполне приемлемым.

Вместе с тем виртуализация добавляет лишний уровень сложности при устранении неполадок. Помимо показателей виртуальной машины, приходится обращать внимание на работоспособность и нагрузку хоста. Что еще хуже, влияние перегруженного хоста на производительность может быть незаметно по показателям в гостевой ОС.

Мы рассмотрим несколько подходов к устранению неполадок на уровне виртуализации в главе 15. Но для начала можно проконсультироваться у специалистов по инфраструктуре, не происходит ли на хосте избыточного резервирования ресурсов. Обратите внимание на количество физических ЦП и выделенных виртуальных ЦП на хосте, а также на физическую и выделенную память. Виртуальным машинам для критически важных экземпляров SQL Server нужно выделять достаточно ресурсов, чтобы их производительность не пострадала.

Если не брать в расчет уровень виртуализации, то на виртуализированных экземплярах SQL Server неполадки устраняются так же, как на обычных. То же самое относится и к облачным конфигурациям SQL Server на виртуальных машинах. В конце концов, облако — это всего лишь особый центр обработки данных, управляемый внешним провайдером.

Настройка SQL-сервера

Конфигурация по умолчанию, применяемая в процессе установки SQL Server, сама по себе неплоха и подходит для легких и даже умеренных нагрузок. Но и в ней есть параметры, которые необходимо проверить и отрегулировать.

Версия SQL Server и уровень обновления

`SELECT @@VERSION` — это первая команда, которую я запускаю во время проверки работоспособности SQL Server. Этому есть две причины. Во-первых, если знать версию, то легче продумывать стратегию отладки системы и предлагать улучшения. Во-вторых, это помогает понять, нет ли в системе уже известных проблем, характерных для этой версии.

Последняя причина очень важна. Клиенты не раз просили меня устранить неполадки, которые уже были устранены в пакетах исправлений и накопительных обновлениях. Всегда просматривайте примечания к выпуску обновлений, потому что может оказаться, что ваша проблема уже решена.

Советую обновляться до новейшей версии SQL Server. В каждой версии улучшаются производительность, функциональность и масштабируемость. Разница особенно заметна, если вы переходите на SQL Server 2016 или более позднюю версию с более старых. Выпуск SQL Server 2016 был важной вехой в истории продукта, и в этой версии появилось множество улучшений, влияющих на производительность. По моему опыту, само по себе обновление с SQL Server 2012 до 2016 или более поздней версии может повысить производительность на 20–40 % без дополнительных усилий.

Стоит также отметить, что, начиная с SQL Server 2016 SP1, многие функции, ранее предназначенные только для Enterprise Edition, появились и в более дешевых версиях. Некоторые из них — например, сжатие данных — позволяют SQL Server кэшировать больше данных в буферном пуле, что повышает производительность.

Очевидно, перед обновлением систему нужно протестировать: всегда существует вероятность, что после обновления она станет работать хуже. В случае небольших патчей такой риск невелик, но с крупными обновлениями лучше быть осторожнее. Некоторые риски можно снизить определенными настройками базы данных, как вы увидите далее в этой главе.

Мгновенная инициализация файлов

Каждый раз, когда SQL Server увеличивает размер файлов или журналов транзакций — будь то автоматически или в рамках команды `ALTER DATABASE`, — он заполняет свежевыделенную часть файла нулями. Этот процесс блокирует все сеансы, которые пытаются записывать в соответствующий файл, а в случае журнала транзакций в нем прекращается создание записей. Также при этом может произойти всплеск нагрузки на систему ввода/вывода.

Для файлов журналов транзакций это поведение нельзя изменить: SQL Server всегда заполняет их нулями. Однако для файлов данных его можно отключить, если активировать мгновенную инициализацию файлов (IFI, instant file

initialization). Она ускоряет разрастание файла данных и сокращает время создания или восстановления баз данных.

Чтобы включить IFI, нужно предоставить стартовой учетной записи SQL Server разрешение `SA_MANAGE_VOLUME_NAME`, также известное как *Perform Volume Maintenance Task* (Выполнить обслуживание томов). Это можно сделать в приложении «Локальная политика безопасности» (*Local Security Policy*, *secpol.msc*). Чтобы изменения вступили в силу, нужно перезапустить SQL Server.

В SQL Server 2016 и более поздних версиях это разрешение также можно предоставить в процессе установки SQL Server, как показано на рис. 1.1.

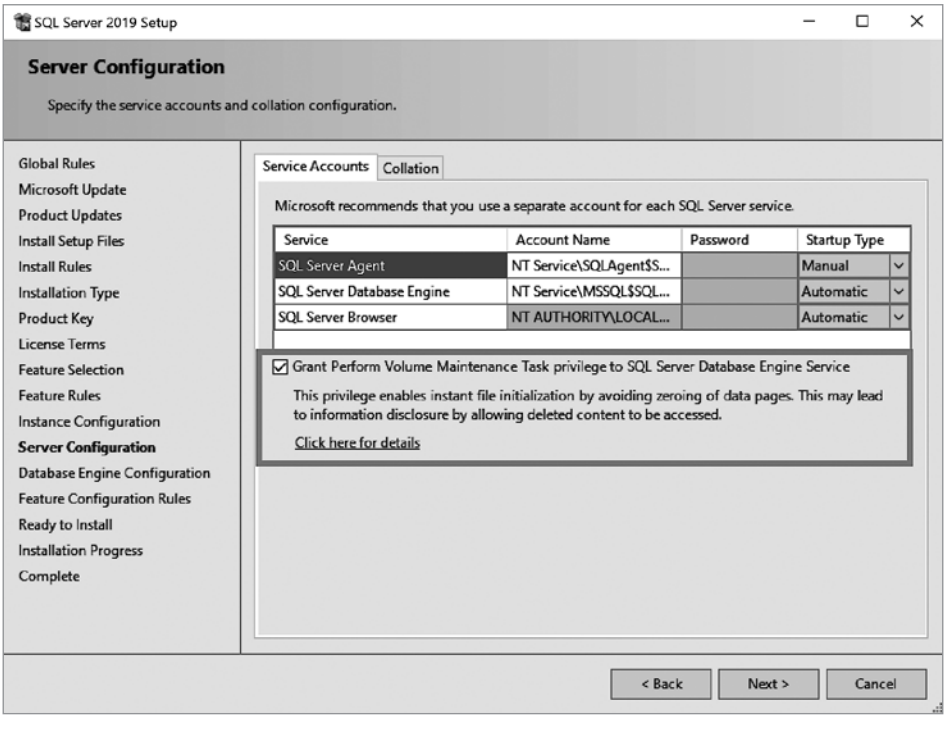


Рис. 1.1. Включение IFI во время установки SQL Server

Чтобы узнать, включена ли IFI, нужно посмотреть на столбец `instant_file_initialization_enabled` в динамическом представлении[1] (DMV) `sys.dm_server_services`. Этот столбец доступен в SQL Server 2012 с пакетом обновления 4 (SP4), SQL Server 2016 с пакетом обновления 1 (SP1) и более поздних версиях. В старых версиях можно запустить код, показанный в листинге 1.1.

[1] https://oreil.ly/58Vd7

Листинг 1.1. Проверка того, включена ли мгновенная инициализация файлов (для старых версий SQL Server)

```
DBCC TRACEON(3004,3605,-1);
GO
CREATE DATABASE Dummy;
GO
EXEC sp_readerrorlog 0,1,N'Dummy';
GO
DROP DATABASE Dummy;
GO
DBCC TRACEOFF(3004,3605,-1);
GO
```

Если IFI не включена, то в журнале SQL Server будет написано, что SQL Server обнуляет файл данных *.mdf* и файл журнала *.ldf* (рис. 1.2). Когда IFI включена, обнуляется только файл журнала *.ldf*.

	LogDate	ProcessInfo	Text
104	2020-10-26 15:57:45.370	spid32s	A connection timeout has occurred while attempting to establish a connection to availa…
105	2020-10-26 15:58:35.510	spid51	DBCC TRACEON 3004, server process ID (SPID) 51. This is an informational message only;…
106	2020-10-26 15:58:35.510	spid51	DBCC TRACEON 3605, server process ID (SPID) 51. This is an informational message only;…
107	2020-10-26 15:58:35.520	spid51	Zeroing C:\DB\Dummy.mdf from page 0 to 1024 (0x0 to 0x800000)
108	2020-10-26 15:58:35.530	spid51	Zeroing completed on C:\DB\Dummy.mdf (elapsed = 2 ms)
109	2020-10-26 15:58:35.530	spid51	Zeroing C:\DB\Dummy_log.ldf from page 0 to 1024 (0x0 to 0x800000)
110	2020-10-26 15:58:35.540	spid51	Zeroing completed on C:\DB\Dummy_log.ldf (elapsed = 2 ms)
111	2020-10-26 15:58:35.550	spid51	Starting up database 'Dummy'.

Рис. 1.2. Проверка настройки мгновенной инициализации файла

С этой настройкой связана небольшая угроза безопасности. Если IFI включена, то администраторы БД могут видеть данные из ранее удаленных файлов в ОС, просматривая свежевыделенные страницы в базе данных. Но для большинства систем это некритично.

Настройка базы tempdb

База `tempdb` — это системная база данных, предназначенная для хранения временных объектов, которые создают пользователи и сам SQL Server. Эта база очень активно используется и часто становится источником состязаний за ресурсы в системе. Как устранять проблемы, связанные с `tempdb`, я расскажу в главе 9, а пока поговорим о настройке.

Как уже упоминалось, базу данных `tempdb` стоит размещать на самом быстром диске. В общем случае этому диску не требуется резервное копирование или другие меры предохранения данных: `tempdb` создается заново при каждом запуске SQL Server, так что для нее вполне подойдет локальный SSD-накопитель или

облачное хранилище. Но помните, что если база данных `tempdb` будет недоступна, то SQL Server перестанет работать.

Если у вас не Enterprise версия SQL Server и в системе больше памяти, чем он потребляет, то можно поместить `tempdb` на RAM-диск. Но с SQL Server Enterprise Edition так поступать не следует: вы добьетесь большей производительности, если используете эту память для буферного пула.

Предварительно выделяйте для файлов `tempdb` место, равное максимальному размеру RAM-диска, и создавайте дополнительные небольшие файлы данных и журналов на диске, чтобы предотвратить нехватку места. SQL Server не будет использовать небольшие файлы на диске, пока RAM-диск не заполнится.

В базе данных `tempdb` всегда должно быть несколько файлов данных. К сожалению, конфигурация по умолчанию, которая создается во время установки SQL Server, неоптимальна, особенно в старых версиях. В главе 9 я расскажу, как точно настроить количество файлов данных в `tempdb`, а пока можно опираться на эмпирические правила:

- Если на сервере восемь или меньше ядер ЦП, создайте такое же количество файлов данных, сколько и ядер.
- Если на сервере больше восьми ядер ЦП, создайте либо восемь файлов данных, либо четверть от числа ядер — в зависимости от того, что больше, — округляя до пакетов по четыре файла. Например, на 24-ядерном сервере нужно 8 файлов данных, а на 40-ядерном — 12 файлов.

Наконец, убедитесь, что у всех файлов данных `tempdb` одинаковый начальный размер и что параметры автоувеличения указаны в мегабайтах, а не в процентах. Это позволит SQL Server сбалансированно использовать файлы данных и уменьшить состязания за использование ресурсов в системе.

Флаги трассировки

Флаги трассировки в SQL Server позволяют активировать некоторые функции или изменить их поведение. В новых версиях SQL Server появляется все больше параметров конфигурации базы данных и сервера, но флаги трассировки по-прежнему широко используются. Вам нужно будет изучить, какие флаги есть в системе, и, возможно, включить некоторые из них.

Чтобы получить список включенных флагов трассировки, выполните команду `DBCC TRACESTATUS`. Флаги можно включить в диспетчере конфигурации SQL Server и/или с помощью параметра `-T` при запуске SQL Server.

Посмотрим на некоторые часто используемые флаги трассировки.

`T1118`

Этот флаг запрещает использовать в SQL Server смешанные экстенты[1]. Это позволяет повысить пропускную способность `tempdb` в SQL Server 2014 и более ранних версиях, потому что уменьшается количество изменений и, следовательно, состязаний за ресурсы в системных каталогах `tempdb`. Этот флаг не нужен в SQL Server 2016 и более поздних версиях, где `tempdb` по умолчанию не использует смешанные экстенты.

`T1117`

Если этот флаг установлен, то SQL Server автоматически увеличивает все файлы данных в файловой группе, когда в одном из файлов заканчивается место. Это позволяет более сбалансированно распределять ввод/вывод по файлам данных. В старых версиях SQL Server этот флаг стоит включить, чтобы улучшить пропускную способность `tempdb`, но лучше проверить, есть ли в базах данных пользователей файловые группы с несколькими файлами данных несбалансированного размера. Как и в случае с `T1118`, этот флаг не нужен в SQL Server 2016 и более поздних версиях, где `tempdb` по умолчанию автоматически увеличивает все файлы данных.

`T2371`

По умолчанию SQL Server автоматически обновляет статистику только после того, как в индексе изменилось 20 % данных. Это означает, что для больших таблиц статистика редко обновляется автоматически. Флаг трассировки `T2371` делает динамическим пороговое значение, при котором обновляется статистика: чем больше таблица, тем меньший процент изменений необходим для обновления статистики. Начиная с SQL Server 2016, это поведение также можно контролировать с помощью уровня совместимости базы данных. Тем не менее я все равно рекомендую включать этот флаг трассировки, если только у всех баз данных на сервере уровень совместимости не составляет 130 или выше.

`T3226`

Когда этот флаг включен, SQL Server не заносит в журнал ошибок записи об успешном создании резервных копий базы данных. Это помогает уменьшить размер журналов, чтобы с ними было удобнее работать.

`T1222`

Этот флаг заносит граф взаимных блокировок в журнал ошибок SQL Server. Он бывает полезен, но читать и анализировать журналы SQL Server становится сложнее. К тому же он избыточен, потому что граф взаимных блоки-

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

ровок при необходимости можно получить из сеанса расширенного события `System_Health`. Я обычно отключаю этот флаг.

`T4199`

Этот флаг и параметр базы данных `QUERY_OPTIMIZER_HOTFIXES` (в SQL Server 2016 и более поздних версиях) управляют поведением исправлений оптимизатора запросов. Если флаг включен, то будут использоваться исправления из пакетов исправлений и накопительных обновлений. Это поможет устранить некоторые ошибки оптимизатора запросов и повысить производительность запросов, но увеличивает риск регрессии планов после исправлений. Обычно я не включаю этот флаг в промышленных экземплярах, если только нет возможности тщательно протестировать систему на предмет регрессий перед тем, как применять исправления.

`T7412`

Этот флаг включает упрощенное профилирование инфраструктуры в SQL Server 2016 и 2017. Он позволяет собирать планы выполнения и множество метрик выполнения запросов, не перегружая ЦП. Я расскажу об этом подробнее в главе 5.

Резюмируем: в SQL Server 2014 и более ранних версиях включайте `T1118`, `T2371` и, возможно, `T1117`. В SQL Server 2016 и более поздних версиях включайте `T2371`, кроме случаев, когда у всех баз данных на сервере уровень совместимости составляет 130 или выше. После этого посмотрите на все остальные флаги трассировки в системе и разберитесь, что они делают. Некоторые флаги устанавливаются без вашего ведома сторонними средствами и могут ухудшить производительность сервера.

Параметры сервера

У SQL Server есть множество параметров конфигурации. Я подробно опишу многие из них позже, но некоторые параметры рассмотрим сейчас.

Оптимизация для нерегламентированной рабочей нагрузки

Первый параметр конфигурации, о котором я расскажу, — *Optimize for Ad-hoc Workloads* (Оптимизировать для нерегламентированной рабочей нагрузки). От него зависит, как SQL Server кэширует планы выполнения нерегламентированных (непараметризованных) запросов. Когда этот параметр отключен (по умолчанию), SQL Server кэширует полные планы выполнения этих инструкций, отчего кэшу планов может понадобиться существенно больше памяти. Когда параметр включен, SQL Server сначала кэширует небольшую структуру (всего несколько сотен байтов) — так называемую *заглушку плана*, — а если запрос выполняется во второй раз, то заменяет заглушку полным планом выполнения.

В большинстве случаев нерегламентированные запросы выполняются однократно, поэтому имеет смысл включить *Optimize for Ad-hoc Workloads*. От этого может значительно сократиться использование памяти кэша планов — правда, изредка нерегламентированные запросы будут дополнительно перекомпилироваться. Очевидно, что этот параметр не влияет на кэширование параметризованных запросов и кода базы данных T-SQL.

Начиная с SQL Server 2019 и баз данных Azure SQL, параметр *Optimize for Ad-hoc Workloads* можно регулировать на уровне базы данных с помощью настройки `OPTIMIZE_FOR_AD_HOC_WORKLOADS`.

Максимальная память сервера

Второй важный параметр — *Max Server Memory*, который определяет, сколько памяти может потреблять SQL Server. Специалисты по базам данных любят спорить о том, как правильно настроить этот параметр, и существуют разные подходы к его вычислению. Многие даже предлагают оставить значение по умолчанию и разрешить SQL Server управлять им автоматически. На мой взгляд, лучше всего настроить его самостоятельно, но делать это нужно грамотно (подробнее в главе 7). Неудачно настроенный параметр может ухудшить быстродействие существеннее, чем значение по умолчанию.

На практике я часто сталкиваюсь с тем, что этому параметру уделяют недостаточно внимания. Иногда его забывают изменить после обновления оборудования или виртуальной машины, а иногда его неправильно рассчитывают в средах, где SQL Server работает на сервере совместно с другими приложениями. В обоих случаях, чтобы повысить производительность, можно для начала просто увеличить параметр *Max Server Memory* или даже перенастроить его на значение по умолчанию, а полноценным анализом заняться позже.

Маска соответствия

Стоит проверить процессорное соответствие SQL Server и, возможно, установить маску соответствия (affinity mask), если SQL Server работает на оборудовании с несколькими узлами неоднородного доступа к памяти (NUMA — non-uniform memory access). В современном аппаратном обеспечении каждый физический ЦП обычно становится отдельным узлом NUMA. Если вы разрешаете SQL Server использовать не все физические ядра, то нужно равномерно распределить процессоры SQL Server (или планировщики — см. главу 2) по NUMA.

Например, если SQL Server работает на сервере с двумя 18-ядерными процессорами Xeon и вы ограничиваете SQL Server до 24 ядер, то нужно установить маску привязки, которая задействует по 12 ядер от каждого физического ЦП. Производительность будет лучше, чем если бы SQL Server задействовал 18 ядер от первого процессора и 6 от второго.

В листинге 1.2 показано, как анализировать распределение планировщиков SQL Server (ЦП) между узлами NUMA. Обратите внимание на количество планировщиков для каждого столбца `parent_node_id` на выходе.

Листинг 1.2. Проверка распределения планировщиков узлов NUMA

```
SELECT
  parent_node_id
  ,COUNT(*) as [Schedulers]
  ,SUM(current_tasks_count) as [Current]
  ,SUM(runnable_tasks_count) as [Runnable]
FROM sys.dm_os_schedulers
WHERE status = 'VISIBLE ONLINE'
GROUP BY parent_node_id;
```

Параллелизм

Важно проверить настройки параллельных операций в системе. Настройки по умолчанию, например `MAXDOP = 0` и `Cost Threshold for Parallelism = 5`, в современных системах работают плохо. Как и в случае с максимальной памятью сервера, лучше подобрать параметры в соответствии с рабочей нагрузкой системы (в главе 6 обсудим это подробно). Могу предложить эмпирическое правило:

- Установите `MAXDOP` равным четверти количества доступных ЦП в OLTP и половине количества доступных ЦП в хранилище данных. На очень больших серверах OLTP оставьте `MAXDOP` равным 16 или ниже. Не превышайте количество планировщиков в узле NUMA.
- `Cost Threshold for Parallelism` установите равным 50.

Начиная с SQL Server 2016, и в серверных базах данных Azure SQL можно установить `MAXDOP` на уровне базы данных с помощью команды `ALTER DATABASE SCOPED CONFIGURATION SET MAXDOP`. Это полезно, когда на одном сервере размещаются базы данных с разными рабочими нагрузками.

Параметры конфигурации

Как и в случае с флагами трассировки, проанализируйте и другие изменения параметров конфигурации, выполненные на сервере. Параметры конфигурации перечислены в представлении `sys.configurations`[1]. К сожалению, в SQL Server нельзя штатными средствами посмотреть параметры, заданные по умолчанию. Чтобы сравнить их с текущими параметрами, придется закодировать соответствующий список, как показано в листинге 1.3. Здесь для экономии места приведено лишь несколько параметров, но из сопутствующих материалов этой книги можно загрузить полную версию сценария.

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

Листинг 1.3. Поиск изменений в настройках конфигурации сервера

```
DECLARE
    @defaults TABLE
    (
        name SYSNAME NOT NULL PRIMARY KEY,
        def_value SQL_VARIANT NOT NULL
    )

INSERT INTO @defaults(name,def_value)
VALUES('backup compression default',0);
INSERT INTO @defaults(name,def_value)
VALUES('cost threshold for parallelism',5);
INSERT INTO @defaults(name,def_value)
VALUES('max degree of parallelism',0);
INSERT INTO @defaults(name,def_value)
VALUES('max server memory (MB)',2147483647);
INSERT INTO @defaults(name,def_value)
VALUES('optimize for ad hoc workloads',0);
/* Прочие параметры опущены в этой книге */

SELECT
    c.name, c.description, c.value_in_use, c.value
    ,d.def_value, c.is_dynamic, c.is_advanced
FROM
    sys.configurations c JOIN @defaults d ON
        c.name = d.name
WHERE
    c.value_in_use <> d.def_value OR
    c.value <> d.def_value
ORDER BY
    c.name;
```

На рис. 1.3 приведен пример вывода предыдущего кода. Если столбцы `value` и `value_in_use` не совпадают, это указывает на заготовленные изменения конфигурации, которые вступят в силу после перезагрузки. Столбец `is_dynamic` показывает, можно ли изменить параметр конфигурации без перезапуска.

	name	description	value_in_use	value	def_value	is_dynamic	is_advanced
1	max degree of parallelism	maximum degree of paralle…	1	1	0	1	1
2	optimize for ad hoc workloads	When this option is set, …	1	1	0	1	1

Рис. 1.3. Измененные параметры конфигурации сервера

Настройка баз данных

На следующем шаге надо проверить некоторые настройки базы данных и параметры конфигурации. Давайте изучим их.

Настройки базы данных

SQL Server позволяет регулировать многие настройки базы данных и управлять ее поведением в зависимости от нагрузки на систему и других требований. О многих из них мы поговорим позже в этой книге, но несколько настроек рассмотрим прямо сейчас.

Первый параметр — *Auto Shrink* (Автоматическое сжатие). Когда он включен, SQL Server периодически сжимает базу данных и возвращает высвободившееся пространство операционной системе. Этот параметр выглядит привлекательно и вроде бы позволяет оптимизировать использование дискового пространства, но он также может вызвать проблемы.

Алгоритм сжатия базы данных работает на физическом уровне. Он находит пустое пространство в начале файла и переносит размещенные экстенты из конца файла в это пустое пространство, не учитывая, кто владелец экстента. Это создает заметную нагрузку и приводит к существенной фрагментации индекса. Более того, во многих случаях сжатие бесполезно: файлы рано или поздно все равно увеличатся снова. Всегда лучше управлять файловым пространством вручную и отключать автоматическое сжатие.

Параметр *Auto Close* (Автоматическая очистка) управляет тем, как SQL Server кэширует данные из базы данных. Когда он включен, SQL Server удаляет страницы данных из буферного пула и планы выполнения из кэша планов, если нет активных подключений к базе. Это снижает производительность новых сеансов, когда данные нужно опять кэшировать, а запросы — компилировать заново.

Как правило, *Auto Close* следует отключать. Но бывают исключения: например, экземпляры, на которых размещено большое количество редко используемых баз данных. Хотя даже в этом случае я бы подумал о том, чтобы оставить эту настройку отключенной и разрешить SQL Server очищать кэш обычным способом.

Убедитесь, что для параметра *Page Verify* (Верификация страниц) установлено значение `CHECKSUM`. Это позволяет эффективнее обнаруживать ошибки согласованности и исправлять повреждения базы данных.

Обратите внимание на модель восстановления базы данных (database recovery model). Если используется режим восстановления `SIMPLE`, то в случае аварии будет невозможно восстановить базу из резервных копий, сделанных позже, чем последняя полная (`FULL`) копия. Если вы обнаружили, что база работает в таком режиме, немедленно обсудите это с заинтересованными сторонами и убедитесь, что они понимают риски потери данных.

Параметр *Database Compatibility Level* (Уровень совместимости БД) управляет совместимостью и поведением SQL Server на уровне базы данных. Например, если вы используете SQL Server 2019 и у вас есть база данных с уровнем совместимости 130 (SQL Server 2016), то SQL Server будет вести себя так, как если бы

база работала на SQL Server 2016. Если держать базы данных на более низких уровнях совместимости, то SQL Server будет проще обновлять, не опасаясь уменьшения производительности. Однако при этом также не будут доступны некоторые новые функции и улучшения.

Как правило, базу данных лучше запускать на последнем уровне совместимости, соответствующем версии SQL Server. Изменяйте уровень с осторожностью, потому что это, как любая смена версии, может снизить производительность. Перед изменениями протестируйте систему и убедитесь, что при необходимости вы сможете откатить изменение, особенно если база данных имеет уровень совместимости 110 (SQL Server 2012) или ниже. На уровне совместимости 120 (SQL Server 2014) или выше включается новая модель оценки количества элементов и могут существенно измениться планы выполнения запросов. Тщательно протестируйте систему, чтобы понять, к чему приведут изменения.

Чтобы SQL Server использовал устаревшие модели оценки количества элементов с новыми уровнями совместимости базы данных, в SQL Server 2016 и более поздних версиях установите для параметра базы данных `LEGACY_CARDINALITY_ESTIMATION` значение `ON`, а в SQL Server 2014 включите флаг трассировки на уровне сервера `T9481`. Этот подход позволит внедрять обновления или менять уровни совместимости поэтапно, сглаживая влияние на систему. (В главе 5 мы подробнее рассмотрим оценку количества элементов и обсудим, как снизить риски при обновлении SQL Server и изменениях уровня совместимости базы данных.)

Настройки журнала транзакций

SQL Server ведет журнал с опережающей записью, сохраняя информацию обо всех изменениях базы данных в журнале транзакций. SQL Server обрабатывает журналы транзакций последовательно, по принципу карусели. В большинстве случаев нет нужды заводить сразу несколько файлов журналов в системе: при этом администрировать базу данных становится сложнее, а производительность не растет.

На внутреннем уровне SQL Server разбивает журналы транзакций на фрагменты, называемые виртуальными файлами журнала (VLF — Virtual Log Files), и управляет ими как цельными единицами. Например, SQL Server не может усечь и повторно использовать VLF, если он содержит только одну активную запись журнала. Следите за количеством VLF в базе данных. При слишком малом количестве очень больших VLF управление журналом и его усечение будут неоптимальными. При слишком большом количестве небольших VLF снизится производительность операций с журналом транзакций. Стремитесь, чтобы в промышленных системах накапливалось не больше нескольких сотен VLF.

Количество VLF, которые SQL Server добавляет при увеличении журнала, зависит от версии SQL Server и размера увеличения. В большинстве случаев создается 8 VLF, если увеличение составляет от 64 Мбайт до 1 Гбайт, или 16 VLF, если увеличение превышает 1 Гбайт. Не следует полагаться на автоматическую настройку, основанную на проценте от увеличения, потому что при этом генерируется множество VLF неравномерного размера. Вместо этого измените параметр автоувеличения журнала, чтобы файл увеличивался пошагово. Я обычно использую шаги по 1024 Мбайт, что дает 128 Мбайт VLF, если только мне не нужен очень большой журнал транзакций.

В SQL Server 2016 и более поздних версиях можно подсчитать число VLF в базе данных с помощью представления `sys.dm_db_log_info`. В более старых версиях SQL Server эту информацию можно получить командой `DBCC LOGINFO`. Если журнал транзакций настроен неправильно, его имеет смысл перестроить. Для этого можно сократить журнал до минимального размера и увеличивать его шагами от 1024 Мбайт до 4096 Мбайт.

Не сжимайте файлы журналов транзакций автоматически. Они снова вырастут и снизят производительность, когда SQL Server обнулит файл. Лучше заранее выделить место и управлять размером файла журнала вручную. Однако не ограничивайте максимальный размер и автоувеличение (autogrowth), иначе журналы не смогут автоматически увеличиваться в случае чрезвычайных ситуаций. (В главе 11 мы подробнее поговорим о том, как устранять проблемы с журналом транзакций.)

Файлы данных и файловые группы

По умолчанию SQL Server создает новые базы данных, используя файловую группу `PRIMARY`, состоящую из одного файла, и один файл журнала транзакций. К сожалению, эта конфигурация неоптимальна с точки зрения производительности, управления базами данных и доступности.

SQL Server отслеживает, как файлы данных используют дисковое пространство, с помощью системных страниц, называемых *картами распределения*. В системах с очень изменчивыми данными карты распределения могут становиться источником состязания за ресурсы: SQL Server обеспечивает строго последовательный доступ к ним во время модификации (подробнее об этом в главе 10). У каждого файла данных есть собственный набор страниц карт распределения, и вы можете уменьшить состязания, если создадите несколько файлов в файловой группе с активно изменяемыми данными.

Убедитесь, что данные равномерно распределены по всем файлам в каждой файловой группе. В SQL Server используется алгоритм пропорционального заполнения, который записывает бóльшую часть данных в файл, в котором больше свободного места. Файлы данных одинакового размера позволяют сба-

лансировать эти операции записи, уменьшив состязания карт распределения. Поэтому проверьте, что у всех файлов данных в файловой группе одинаковый размер и шаг автоматического увеличения, указанный в мегабайтах.

Также вы можете включить параметр файловой группы `AUTOGROW_ALL_FILES` (доступен в SQL Server 2016 и более поздних версиях), который запускает автоувеличение для всех файлов в файловой группе одновременно. В предыдущих версиях SQL Server для этого можно использовать флаг трассировки `T1117`, однако имейте в виду, что он устанавливается на уровне сервера и влияет на все базы данных и файловые группы в системе.

Изменять структуру существующих баз данных обычно нецелесообразно или невозможно. Но может потребоваться создавать новые файловые группы и перемещать данные, чтобы оптимизировать производительность. Приведем несколько советов, как делать это эффективно:

- Создайте несколько файлов данных в файловых группах с изменчивыми данными. Обычно я начинаю с четырех файлов и увеличиваю их количество, если вижу проблемы с кратковременной блокировкой (см. главу 10). Убедитесь, что у всех файлов данных одинаковый размер и параметры автоувеличения; включите параметр `AUTOGROW_ALL_FILES`. Для файловых групп, данные в которых предназначены только для чтения, обычно достаточно одного файла данных.
- Не разбивайте кластеризованные индексы, некластеризованные индексы или большие объекты (LOB) по разным файловым группам. Это редко помогает повысить производительность, зато может привести к проблемам в случае повреждения базы данных.
- Помещайте связанные сущности (например, `Orders` и `OrderLineItems`) в одну и ту же файловую группу. Это упростит управление базой данных и аварийное восстановление.
- По возможности оставляйте пустой файловую группу `PRIMARY`.

На рис 1.4 показан пример структуры базы данных для гипотетической системы электронной коммерции. Данные разбиты на секции и распределены по нескольким файловым группам, чтобы свести к минимуму время простоя и получить возможность использовать хотя бы часть базы данных в случае аварии[1]. Это также позволяет улучшить стратегию резервного копирования: можно копировать базу данных по частям и исключить из полных резервных копий данные, предназначенные только для чтения.

[1] Чтобы глубже погрузиться в стратегии разбиения данных и аварийного восстановления, читайте мою книгу «Pro SQL Server Internals, Second Edition» (Apress, 2016).

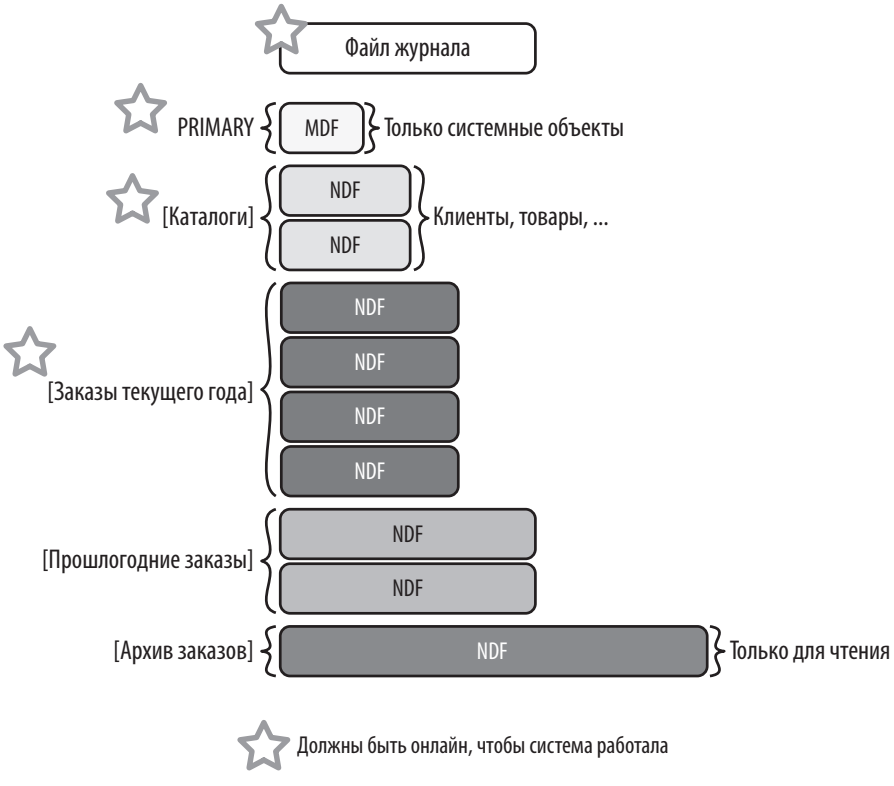


Рис. 1.4. Структура базы данных для системы электронной коммерции

Анализ журнала ошибок SQL Server

Журнал ошибок SQL Server — еще одно место, куда я обязательно заглядываю в начале устранения неполадок. Ошибки, зафиксированные в этом журнале, часто указывают на конкретные очаги проблем. Например, ошибки 823 и 824 могут свидетельствовать о проблемах с дисковой подсистемой и/или повреждением базы данных.

Просмотреть содержимое журнала ошибок можно средствами SSMS. Его также можно получить программно с помощью системной хранимой процедуры `xp_readerrorlog`. Проблема здесь заключается в количестве данных в журнале: полезные данные могут затеряться среди множества информационных сообщений.

Код в листинге 1.4 помогает решить эту проблему. Он отфильтровывает ненужный шум и позволяет сосредоточиться на сообщениях об ошибках. Управлять поведением кода можно с помощью следующих переменных:

`@StartDate` и `@EndDate`

Задают диапазон времени для анализа.

`@NumErrorLogs`

Указывает количество файлов журналов для чтения, если SQL Server переключается на файлы продолжения.

`@ExcludeLogonErrors`

Опускает сообщения аудита входа в систему.

`@ShowSurroundingEvents` и `@ExcludeLogonSurroundingEvents`

Позволяют получать информационные сообщения в ближайшей окрестности записей об ошибках из журнала. Временно́е окно для этих сообщений управляется переменными `@SurroundingEventsBeforeSeconds` и `@SurroundingEventsAfterSeconds`.

Сценарий выдает два результата. Первый — записи из журнала ошибок, которые содержат слово *error*. Когда параметр `@ShowSurroundingEvents` включен, сценарий также выводит записи журнала в ближайшей окрестности этих error-строк. Некоторые записи, содержащие слово *error*, можно исключить из вывода, если вставить их в таблицу `@ErrorsToIgnore`.

Листинг 1.4. Анализ журнала ошибок SQL Server

```
IF OBJECT_ID('tempdb..#Logs',N'U') IS NOT NULL DROP TABLE #Logs;
IF OBJECT_ID('tempdb..#Errors',N'U') IS NOT NULL DROP TABLE #Errors;
GO

CREATE TABLE #Errors
(
  LogNum INT NULL,
  LogDate DATETIME NULL,
  ID INT NOT NULL identity(1,1),
  ProcessInfo VARCHAR(50) NULL,
  [Text] NVARCHAR(MAX) NULL,
  PRIMARY KEY(ID)
);

CREATE TABLE #Logs
(
  [LogDate] DATETIME NULL,
  ProcessInfo VARCHAR(50) NULL,
  [Text] NVARCHAR(MAX) NULL
);

DECLARE
  @StartDate DATETIME = DATEADD(DAY,-7,GETDATE())
  ,@EndDate DATETIME = GETDATE()
  ,@NumErrorLogs INT = 1
```

```
  ,@ExcludeLogonErrors BIT = 1
  ,@ShowSurroundingEvents BIT = 1
  ,@ExcludeLogonSurroundingEvents BIT = 1
  ,@SurroundingEventsBeforeSecond INT = 5
  ,@SurroundingEventsAfterSecond INT = 5
  ,@LogNum INT = 0;

DECLARE
  @ErrorsToIgnore TABLE
  (
    ErrorText NVARCHAR(1024) NOT NULL
  );

INSERT INTO @ErrorsToIgnore(ErrorText)
VALUES
  (N'Registry startup parameters:%'),
  (N'Logging SQL Server messages in file%'),
  (N'CHECKDB for database%finished without errors%');

WHILE (@LogNum <= @NumErrorLogs)
BEGIN
  INSERT INTO #Errors(LogDate,ProcessInfo,Text)
    EXEC [master].[dbo].[xp_readerrorlog]
      @LogNum, 1, N'error', NULL, @StartDate, @EndDate, N'desc';
  IF @@ROWCOUNT > 0
    UPDATE #Errors SET LogNum = @LogNum WHERE LogNum IS NULL;
  SET @LogNum += 1;
END;

IF @ExcludeLogonErrors = 1
  DELETE FROM #Errors WHERE ProcessInfo = 'Logon';

DELETE FROM e
FROM #Errors e
WHERE EXISTS
(
  SELECT *
  FROM @ErrorsToIgnore i
  WHERE e.Text LIKE i.ErrorText
);

-- Только ошибки
SELECT * FROM #Errors ORDER BY LogDate DESC;

IF @@ROWCOUNT > 0 AND @ShowSurroundingEvents = 1
BEGIN
  DECLARE
    @LogDate DATETIME
    ,@ID INT = 0

  WHILE 1 = 1
  BEGIN
    SELECT TOP 1 @LogNum = LogNum, @LogDate = LogDate, @ID = ID
    FROM #Errors
```

```
    WHERE ID > @ID
    ORDER BY ID;

    IF @@ROWCOUNT = 0
      BREAK;

    SELECT
      @StartDate = DATEADD(SECOND, -@SurroundingEventsBeforeSecond, @LogDate)
      ,@EndDate = DATEADD(SECONd, @SurroundingEventsAfterSecond, @LogDate);

    INSERT INTO #Logs(LogDate,ProcessInfo,Text)
      EXEC [master].[dbo].[xp_readerrorlog]
        @LogNum, 1, NULL, NULL, @StartDate, @EndDate;

  END;

  IF @ExcludeLogonSurroundingEvents = 1
    DELETE FROM #Logs WHERE ProcessInfo = ,Logon';

  DELETE FROM e
  FROM #Logs e
  WHERE EXISTS
    (
    SELECT *
    FROM @ErrorsToIgnore i
    WHERE e.Text LIKE i.ErrorText
  );

  SELECT * FROM #Logs ORDER BY LogDate DESC;
END
```

Я не привожу здесь полный список возможных ошибок, потому что он может быть очень длинным и во многих случаях зависит от конкретной системы. Но вам стоит проанализировать любые подозрительные результаты на выходе этого сценария и разобраться, как они могут повлиять на систему.

Наконец, я предлагаю настроить оповещения об ошибках с высоким уровнем серьезности в SQL Server Agent, если вы еще их не настроили. Как это сделать, можно узнать из документации Microsoft[1].

Консолидация экземпляров и баз данных

Невозможно говорить об устранении неполадок SQL Server, не затронув вопросы консолидации баз данных и экземпляров SQL Server. Консолидация часто снижает затраты на оборудование и лицензии, но она тоже имеет свою цену, и вам нужно проанализировать, как консолидация может ухудшить текущую или будущую производительность системы.

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

Не существует универсальной стратегии консолидации, которая подошла бы к любому проекту. Принимая решение о консолидации, нужно учесть объем данных, нагрузку, конфигурацию оборудования, а также требования бизнеса и безопасности. Однако в общем случае следует избегать консолидации баз данных OLTP и хранилища данных/отчетов на одном сервере, если они работают под большой нагрузкой. (А если они уже консолидированы, их стоит разделить.) Запросы к хранилищу данных обычно оперируют большими объемами данных, что приводит к интенсивному вводу/выводу и сбросу содержимого буферного пула. В совокупности это негативно сказывается на производительности других систем.

Кроме того, при консолидации баз данных следует проанализировать требования к безопасности. Некоторые функции безопасности, такие как аудит, затрагивают весь сервер и ограничивают производительность всех баз данных на сервере. Еще один пример — прозрачное шифрование данных (TDE, Transparent Data Encryption). Хотя TDE — функция уровня базы данных, тем не менее SQL Server шифрует `tempdb`, если TDE включено хотя бы на одной базе. Это ограничивает производительность всех остальных систем.

Как правило, на одном и том же экземпляре SQL Server не стоит хранить базы данных с разными требованиями к безопасности. Следует проанализировать тенденции и выбросы в показателях и при необходимости отделить базы друг от друга. (Позже в этой книге я покажу код, который поможет проанализировать использование ЦП, операции ввода/вывода и затраты памяти для каждой базы данных.)

Я предлагаю использовать виртуализацию и консолидировать несколько виртуальных машин на одном или нескольких хостах вместо того, чтобы размещать несколько независимых и активных баз данных на одном экземпляре SQL Server. Такой подход обеспечивает гораздо бóльшую гибкость, управляемость и взаимную изоляцию систем, особенно если несколько экземпляров SQL Server работают на одном сервере. Используя виртуализацию, гораздо проще контролировать потребление ими ресурсов.

Эффект наблюдателя

Для промышленного развертывания всякой серьезной системы SQL Server нужно внедрить стратегию мониторинга. Это могут быть либо сторонние средства мониторинга, либо код, созданный на основе стандартных технологий SQL Server, либо и то и другое.

Хорошая стратегия мониторинга жизненно важна для полноценной поддержки SQL Server. Мониторинг позволяет действовать на упреждение, сокращая количество инцидентов и время восстановления работоспособности. К сожалению,

даром ничего не дается, и любой мониторинг увеличивает нагрузку на систему. В некоторых случаях эти накладные расходы оказываются незначительными и приемлемыми, но иногда они существенно влияют на производительность сервера.

За свою карьеру консультанта по SQL Server я видел множество примеров неэффективного мониторинга. Например, один клиент использовал инструмент, который предоставлял информацию о фрагментации индекса, вызывая функцию `sys.dm_db_index_physical_stats` в режиме `DETAILED` каждые четыре часа для каждого индекса в базе данных. Это приводило к огромным пикам ввода/вывода и очистке буферного пула, что сильно било по производительности. Другой клиент применял инструмент, который постоянно опрашивал различные DMV, значительно увеличивая нагрузку ЦП на сервере.

К счастью, во время устранения неполадок в системе такие запросы часто удается проанализировать, чтобы оценить их влияние. Но с другими технологиями это не всегда получается. Пример такого случая — мониторинг на основе расширенных событий (xEvents — Extended Events). Extended Events — это отличная технология, которая позволяет устранять сложные проблемы в SQL Server, но она плохо подходит в качестве инструмента профилирования. Некоторые события весьма тяжеловесны и ведут к большим накладным расходам в загруженных средах.

Рассмотрим пример кода, который создает сеанс расширенных событий. Этот сеанс фиксирует запросы, выполняемые в системе, см. листинг 1.5.

Листинг 1.5. Создание сеанса расширенных событий для захвата запросов в системе

```
CREATE EVENT SESSION CaptureQueries ON SERVER
ADD EVENT sqlserver.rpc_completed
(
  SET collect_statement=(1)
  ACTION
  (
    sqlos.task_time
    ,sqlserver.client_app_name
    ,sqlserver.client_hostname
    ,sqlserver.database_name
    ,sqlserver.nt_username
    ,sqlserver.sql_text
  )
),
ADD EVENT sqlserver.sql_batch_completed
(
  ACTION
  (
    sqlos.task_time
    ,sqlserver.client_app_name
    ,sqlserver.client_hostname
```

```
    ,sqlserver.database_name
    ,sqlserver.nt_username
    ,sqlserver.sql_text
  )
),
ADD EVENT sqlserver.sql_statement_completed
ADD TARGET package0.event_file
(SET FILENAME=N'C:\PerfLogs\LongSql.xel',MAX_FILE_SIZE=(200))
WITH
(
  MAX_MEMORY =4096 KB
  ,EVENT_RETENTION_MODE=ALLOW_SINGLE_EVENT_LOSS
  ,MAX_DISPATCH_LATENCY=5 SECONDS
);
```

Попробуйте выполнить этот код на сервере, который работает под большой нагрузкой с большим количеством одновременных запросов. Измерьте пропускную способность системы с запущенным сеансом расширенных событий и без него. Конечно же, будьте осторожны и не запускайте код на рабочем сервере!

На рис. 1.5 показана загрузка ЦП и количество пакетных запросов в секунду в обоих сценариях на одном из моих серверов. Как видите, включение сеанса расширенных событий уменьшило пропускную способность примерно на 20 %. Что еще хуже, обнаружить само существование этого сеанса на сервере довольно сложно.

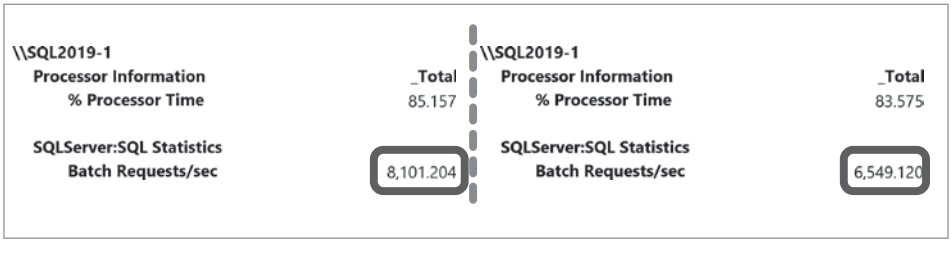


Рис. 1.5. Пропускная способность сервера с активным сеансом расширенных событий и без него

Очевидно, степень воздействия будет зависеть от рабочей нагрузки системы. В любом случае, при устранении неполадок следует проверить, нет ли в системе ненужных инструментов мониторинга или сбора данных.

В общем, оценивайте стратегию мониторинга и, в частности, ее накладные расходы, особенно когда на сервере размещено несколько баз данных. Например, расширенные события работают на уровне сервера. Хотя можно фильтровать события на основе поля `database_id`, эта фильтрация происходит *после* запуска события, так что расширенные события все равно влияют на все базы данных на сервере.

Резюме

Устранение неполадок в системе — это целостный процесс, требующий анализа всей экосистемы. Нужно проанализировать аппаратное обеспечение, ОС и уровни виртуализации, а также настройку SQL Server и баз данных.

У SQL Server есть множество параметров, которые позволяют тонко подстроить функционирование сервера под рабочие нагрузки системы. Существуют общие рекомендации, применимые к большинству систем: например, включить IFI и оптимизацию для нерегламентированных рабочих нагрузок, увеличить количество файлов в базе данных `tempdb`, включить определенные флаги трассировки, отключить автоматическое сжатие и настроить правильные параметры автоувеличения файлов базы данных.

В следующей главе я расскажу об одном из наиболее важных компонентов SQL Server под названием SQLOS и устранении неполадок с помощью статистики ожидания.

Чек-лист устранения неполадок

- ☐ Выполнить высокоуровневый анализ оборудования, сети и дисковой подсистемы.
- ☐ В виртуализированных средах — обсудить со специалистами по инфраструктуре конфигурацию хоста и нагрузку.
- ☐ Изучить версии ОС и SQL Server, номер выпуска и установленные исправления.
- ☐ Проверить, включена ли мгновенная инициализация файла.
- ☐ Проанализировать флаги трассировки.
- ☐ Включить оптимизацию для нерегламентированных рабочих нагрузок.
- ☐ Проверить настройки памяти и параллелизма на сервере.
- ☐ Изучить настройки `tempdb` (включая количество файлов); в версиях SQL Server до 2016 года проверить флаг трассировки `T1118` и, возможно, `T1117`.
- ☐ Отключить автоматическое сжатие баз данных.
- ☐ Проверить настройки файлов журнала данных и транзакций.
- ☐ Проверить число VLF в файлах журнала транзакций.
- ☐ Проверить ошибки в журнале SQL Server.
- ☐ Проверить наличие ненужного мониторинга в системе.

ГЛАВА 2

Модель выполнения SQL Server и статистика ожидания

Невозможно устранять неполадки в SQL Server, не разбираясь в его модели выполнения. Чтобы обнаруживать узкие места в системе, нужно знать, как SQL Server выполняет задачи и управляет ресурсами. Этим темам посвящена данная глава.

В начале главы пойдет речь об архитектуре и основных компонентах SQL Server. Затем мы изучим модель выполнения SQL Server и представим популярный метод устранения неполадок, который называется статистикой ожидания. Также рассмотрим несколько динамических административных представлений, которые обычно используются при устранении неполадок. Завершит главу обзор регулятора ресурсов, с помощью которого можно разделять рабочие нагрузки в системе.

SQL Server: высокоуровневая архитектура

Как известно, SQL Server — это очень сложный продукт, состоящий из десятков компонентов и подсистем, которые невозможно полноценно охватить в одной книге. В этом разделе приведем их обзор на самом общем уровне. Для удобства я разделю эти компоненты и подсистемы на несколько категорий, как показано на рис. 2.1. Давайте поговорим о них.

Уровень протокола обеспечивает связь между SQL Server и клиентскими приложениями. Этот уровень использует внутренний формат *Tabular Data Stream* (TDS, Поток табличных данных), чтобы передавать данные через сетевые протоколы, такие как TCP/IP или именованные каналы. Если клиентское приложение и SQL Server работают на одном компьютере, можно использовать другой протокол — *Shared Memory* (Общая память).

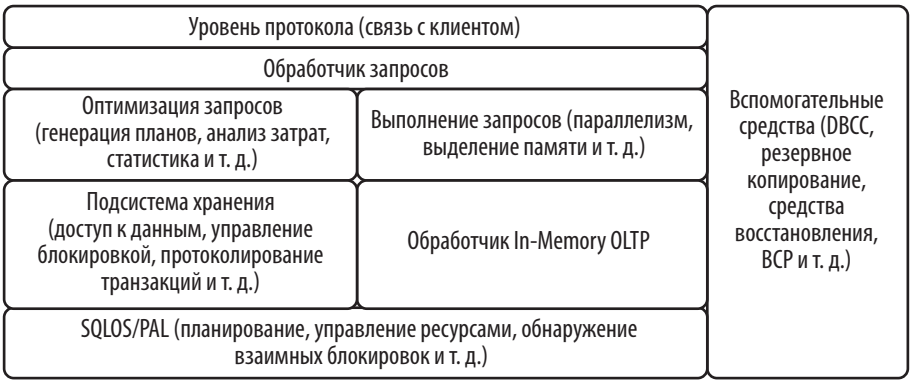


Рис. 2.1. Основные компоненты SQL Server

Устраняя проблемы, касающиеся соединения между клиентом и сервером, стоит проверить, какие протоколы включены. В некоторых версиях SQL Server, например Express и Developer, TCP/IP и именованные каналы по умолчанию отключены, так что сервер не принимает удаленных клиентских подключений, пока вы не включите сетевые протоколы в Диспетчере конфигурации (Configuration Manager) SQL Server.

Уровень обработчика запросов (Query Processor) отвечает за оптимизацию и выполнение запросов. Он осуществляет синтаксический анализ, оптимизирует запросы и управляет скомпилированными планами запросов, а также координирует все аспекты выполнения запросов.

Подсистема хранения (Storage Engine) отвечает за доступ к данным и их обработку в SQL Server. Она взаимодействует с данными на диске, управляет журналами транзакций и обрабатывает транзакции, блокировки и конкурентный доступ, а также выполняет некоторые другие задачи.

Обработчик In-Memory OLTP (In-Memory OLTP Engine) поддерживает In-Memory OLTP в SQL Server. Он работает с таблицами, оптимизированными для памяти, и отвечает за управление данными и доступ к этим таблицам, компиляцию в собственном коде, сохраняемость данных и прочие аспекты этой технологии.

Между компонентами существуют уровни абстракции. Например, *Взаимодействие с запросами (Query Interop)* (не показанное на рис. 2.1) позволяет обработчику запросов работать как с таблицами на основе строк, так и с таблицами, оптимизированными для памяти: запросы прозрачно перенаправляются либо в хранилище, либо в обработчик In-Memory OLTP.

Самый важный уровень абстракции — операционная система SQL Server (SQLOS), которая изолирует другие компоненты SQL Server от нижележащих операционных систем и занимается планированием, управлением ресурсами

и их мониторингом, обработкой исключений и многими другими аспектами работы SQL Server. Например, когда для какого-нибудь компонента SQL Server требуется выделить память, он не вызывает функции API ОС напрямую, а запрашивает память у SQLOS. Это позволяет SQL Server детально контролировать выполнение задач и использование внутренних ресурсов, не полагаясь на ОС.

Наконец, с появлением поддержки Linux в SQL Server 2017 возник еще один компонент под названием *Platform Abstraction Layer* (PAL, *Уровень платформенной абстракции*) — прослойка между SQLOS и операционными системами. За исключением отдельных ситуаций, критичных для производительности, SQLOS не обращается напрямую к API ОС, а работает через PAL. При этом код самого SQL Server под Linux практически идентичен коду под Windows, что значительно ускоряет разработку и усовершенствование продукта.

С точки зрения устранения неполадок разница между SQL Server в Windows и Linux невелика. Конечно, приходится по-разному анализировать экосистему SQL Server и конфигурацию ОС. Но когда дело касается проблем *внутри* SQL Server, обе платформы ведут себя одинаково, поэтому в этой книге я не буду останавливаться на различиях между ними.

Давайте подробнее поговорим о SQLOS.

SQLOS и модель выполнения

Серверы баз данных должны обрабатывать большое количество пользовательских запросов, и SQL Server — не исключение. На верхнем уровне он распределяет запросы по отдельным потокам, чтобы запросы выполнялись одновременно. Если сервер не простаивает, количество активных потоков больше, чем количество ЦП в системе. Эффективное планирование — залог высокой производительности сервера.

В ранних версиях SQL Server использовался планировщик Windows. К сожалению, Windows (и Linux) — это операционные системы общего назначения, а значит, в них используется вытесняющее планирование. Это значит, что ОС выделяет на выполнение потока определенный временной интервал, или *квант времени*, а когда он истекает, переключается на другие потоки. Это затратная операция: она требует переключаться между пользовательским режимом и режимом ядра, что отрицательно сказывается на производительности.

В SQL Server 7.0 Microsoft представила первую версию планировщика пользовательского режима (UMS, User Mode Scheduler): это был тонкий слой между Windows и SQL Server, отвечающий за планирование. В нем использовалось *совместное планирование*: потоки SQL Server были запрограммированы так, чтобы добровольно уступать управление каждые 4 мс, давая другим потокам

возможность поработать. Такой подход значительно сократил затраты на переключение режимов.

Отдельные процессы SQL Server, такие как расширенные хранимые процедуры, подпрограммы CLR, внешние языки и некоторые другие процессы, по-прежнему могут выполняться в режиме вытесняющего планирования.

Microsoft продолжила совершенствовать UMS в SQL Server 2000, а в SQL Server 2005 переработала его, создав более надежную систему SQLOS. В последующих версиях SQL Server SQLOS стала отвечать за планирование, управление памятью и вводом/выводом, обработку исключений, интеграцию с CLR и внешними языками, а также некоторые другие задачи.

Когда вы запускаете процесс SQL Server, SQLOS создает набор планировщиков, которые распределяют рабочую нагрузку между процессорами. Количество планировщиков совпадает с количеством логических ЦП в системе, и еще один планировщик создается для выделенного административного соединения (DAC, Dedicated Admin Connection). Например, если у вас два четырехъядерных физических процессора с гиперпоточностью, то SQL Server создаст 17 планировщиков. На практике можно считать, что планировщик — то же самое, что процессор, и я буду использовать эти термины взаимозаменяемо на протяжении всей книги.

DAC — это *крайняя мера* устранения неполадок подключения. Оно позволяет получить доступ к SQL Server, если он перестал отвечать и не принимает обычные соединения. Я расскажу об этом подробнее в главе 13.

Каждый планировщик может находиться в состоянии `ONLINE` или `OFFLINE` в зависимости от настроек маски соответствия и модели лицензирования на основе ядер. Планировщики обычно не перемещаются с одного процессора на другой, хотя такая миграция возможна, особенно при большой нагрузке. Чаще всего это не влияет на процесс устранения неполадок.

Планировщики управляют набором рабочих потоков, которые также называются *исполнителями (workers)*. Максимальное количество исполнителей в системе определяется параметром конфигурации *Max Worker Thread*. Значение по умолчанию равно нулю, и при этом SQL Server вычисляет максимальное количество исполнителей, исходя из количества планировщиков в системе. Обычно менять значение по умолчанию не требуется да и не рекомендуется, если только вы не знаете *точно*, что делаете.

Каждый раз, когда появляется задача, она назначается свободному исполнителю. Если таких исполнителей нет, планировщик создает нового. Он также уничтожает простаивающих исполнителей через 15 минут бездействия или когда не хватает

памяти. Каждый исполнитель занимает область в памяти, выделенной для стека потоков: 512 Кбайт в 32-разрядной версии и 2 Мбайт в 64-разрядной версии SQL Server. Исполнителей можно рассматривать как логическое представление потоков в ОС, а задачи — как единицы работы, выполняемые этими потоками.

Исполнители не перемещаются от одного планировщика к другому, а задачи не перемещаются между исполнителями. Однако SQLOS может создавать дочерние задачи и назначать их разным исполнителям, например в случае параллельного плана выполнения. Это объясняет ситуации, когда некоторые планировщики нагружены сильнее других: отдельные рабочие процессы могут время от времени получать более ресурсоемкие задачи.

По умолчанию SQL Server назначает задачи исполнителям, обходя узлы NUMA в циклическом режиме и не учитывая количество планировщиков в этих узлах. Неравномерное распределение планировщиков по узлам NUMA приводит к тому, что работа рассредоточивается между планировщиками несбалансированно (я покажу пример такой ситуации в главе 15).

Чаще всего при устранении неполадок мы исследуем именно задачи. Но бывают исключения, когда задача находится в состоянии `PENDING`, то есть она создана и ожидает доступного исполнителя. Это нормальное явление, и обычно исполнители быстро забирают задачи. Однако это также может сигнализировать о весьма опасном состоянии, когда в системе не хватает рабочих процессов для обработки запросов. В главе 13 я расскажу, как обнаруживать и решать эту проблему.

Помимо `PENDING`, задача может находиться в пяти других состояниях:

`RUNNING`

Задача в данный момент выполняется в планировщике.

`RUNNABLE`

Задача ожидает, пока планировщик ее запустит.

`SUSPENDED`

Задача ожидает внешнего события или ресурса.

`SPINLOOP`

Задача обрабатывает спин-блокировку. Спин-блокировки — это объекты синхронизации, служащие для защиты некоторых внутренних объектов. SQL Server может использовать спин-блокировки, когда предполагается, что доступ к объекту будет предоставлен очень быстро, без переключения контекста для исполнителей.

`DONE`

Задача завершена.

Первые три состояния — самые важные и распространенные. У каждого планировщика единовременно может быть не более одной задачи в состоянии RUNNING. Кроме того, у него есть две разные очереди: одна — для задач в состоянии RUNNABLE и одна — для задач в состоянии SUSPENDED. Когда задаче в состоянии RUNNING требуются ресурсы — например, страница данных с диска, — она подает запрос ввода/вывода и переходит в состояние SUSPENDED. Задача находится в очереди SUSPENDED до тех пор, пока запрос не будет выполнен и страница не будет прочитана. После этого, когда задача готова выполняться дальше, она перемещается в очередь RUNNABLE.

Возможно, ближайшая аналогия этого процесса из реальной жизни — общая очередь на кассовом узле в магазине. Здесь кассиры — это планировщики, а покупатели — задачи в очереди RUNNABLE. Покупатель, которого в данный момент обслуживают на кассе, аналогичен задаче в состоянии RUNNING.

Если на товаре нет штрихкода, кассир посылает работника магазина проверить цену. Кассир приостанавливает обслуживание текущего покупателя и просит его отойти в сторону (в очередь SUSPENDED). Когда работник возвращается с информацией о цене, покупатель переходит в конец очереди на кассе (то есть конец очереди RUNNABLE).

Конечно, на SQL Server эти процедуры выполняются намного эффективнее, чем в реальном магазине, где клиентам приходится терпеливо ждать в стороне, пока проверяют цену. (Наверное, покупатель в конце очереди RUNNABLE мечтал бы о таком же быстродействии, как в SQL Server!)

Статистика ожидания

Если не считать процедур инициализации и утилизации, задача на протяжении своего жизненного цикла переключается между состояниями RUNNING, SUSPENDED и RUNNABLE, как показано на рис. 2.2. Общее время выполнения складывается из трех компонентов: время в состоянии RUNNING, когда задача фактически выполняется, время в состоянии RUNNABLE, когда задача ожидает, пока планировщик (ЦП) ее запустит, и время в состоянии SUSPENDED, когда задача ожидает ресурсов.

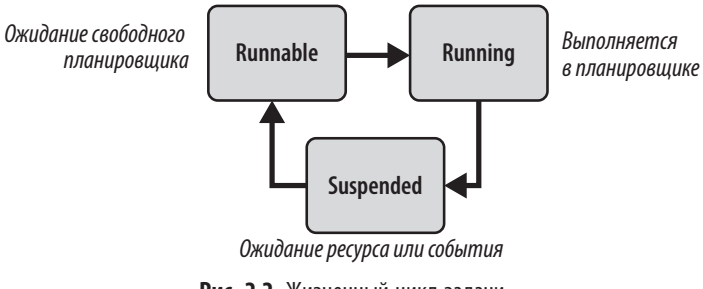


Рис. 2.2. Жизненный цикл задачи

В целом главная задача настройки производительности — в том, чтобы повысить пропускную способность системы, сократив время выполнения запросов. Этого можно добиться, если уменьшить время нахождения задач запросов в любом из трех перечисленных состояний.

Время запроса в состоянии `RUNNING` можно сократить, если модернизировать аппаратное обеспечение и перейти на более быстрые ЦП, а также если уменьшить объем задач благодаря оптимизации запросов. Чтобы сократить время `RUNNABLE`, можно добавить больше ресурсов ЦП или снизить нагрузку на систему. Но наибольшей экономии обычно удается достичь, если оптимизировать время, когда задача находится в состоянии `SUSPENDED`, ожидая ресурсов.

SQL Server отслеживает совокупное время, которое задачи проводят в состоянии `SUSPENDED` для различных типов ожидания. Эти данные можно просмотреть в представлении `sys.dm_os_wait_stats`[1], чтобы получить общую картину узких мест в системе и уточнить стратегию устранения неполадок.

В коде листинга 2.1 показаны типы ожидания, которые занимают больше всего времени. (Опущены некоторые безобидные типы, в основном связанные с внутренними процессами SQL Server, которые проводят бóльшую часть времени в ожидании.) Данные собираются с момента последнего перезапуска SQL Server или с момента последней очистки с помощью команды `DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR)`. В каждой новой версии SQL Server появляются новые типы ожидания. Одни из них полезны для устранения неполадок, а другие не имеет смысл учитывать[2].

Листинг 2.1. Получение основных типов ожидания в системе (SQL Server 2012 и более поздние версии)

```
;WITH Waits
AS
(
  SELECT
    wait_type, wait_time_ms, waiting_tasks_count,signal_wait_time_ms
    ,wait_time_ms - signal_wait_time_ms AS resource_wait_time_ms
    ,100. * wait_time_ms / SUM(wait_time_ms) OVER() AS Pct
    ,100. * SUM(wait_time_ms) OVER(ORDER BY wait_time_ms DESC) /
        NULLIF(SUM(wait_time_ms) OVER(), 0) AS RunningPct
    ,ROW_NUMBER() OVER(ORDER BY wait_time_ms DESC) AS RowNum
  FROM sys.dm_os_wait_stats WITH (NOLOCK)
  WHERE
    wait_type NOT IN /* Исключаем не интересующие нас системные ожидания */
    (N'BROKER_EVENTHANDLER',N'BROKER_RECEIVE_WAITFOR',N'BROKER_TASK_STOP'
```

[1] https://oreil.ly/Nh4s2

[2] Код, приведенный в листинге 2.1, подходит для версий SQL Server 2019 и более старых. Чтобы исключить другие типы ожидания в будущих версиях, обратитесь к документации Microsoft: https://oreil.ly/O4tzq.

```
    ,N'BROKER_TO_FLUSH',N'BROKER_TRANSMITTER',N'CHECKPOINT_QUEUE',N'CHKPT'
    ,N'CLR_SEMAPHORE',N'CLR_AUTO_EVENT',N'CLR_MANUAL_EVENT'
    ,N'DBMIRROR_DBM_EVENT',N'DBMIRROR_EVENTS_QUEUE',N'DBMIRROR_WORKER_QUEUE'
    ,N'DBMIRRORING_CMD',N'DIRTY_PAGE_POLL',N'DISPATCHER_QUEUE_SEMAPHORE'
    ,N'EXECSYNC',N'FSAGENT',N'FT_IFTS_SCHEDULER_IDLE_WAIT',N'FT_IFTSHC_MUTEX'
    ,N'HADR_CLUSAPI_CALL',N'HADR_FILESTREAM_IOMGR_IOCOMPLETION'
    ,N'HADR_LOGCAPTURE_WAIT',N'HADR_NOTIFICATION_DEQUEUE'
    ,N'HADR_TIMER_TASK',N'HADR_WORK_QUEUE',N'KSOURCE_WAKEUP',N'LAZYWRITER_SLEEP'
    ,N'LOGMGR_QUEUE',N'ONDEMAND_TASK_QUEUE'
    ,N'PARALLEL_REDO_WORKER_WAIT_WORK',N'PARALLEL_REDO_DRAIN_WORKER'
    ,N'PARALLEL_REDO_LOG_CACHE',N'PARALLEL_REDO_TRAN_LIST'
    ,N'PARALLEL_REDO_WORKER_SYNC',N'PREEMPTIVE_SP_SERVER_DIAGNOSTICS'
    ,N'PREEMPTIVE_OS_LIBRARYOPS',N'PREEMPTIVE_OS_COMOPS', N'PREEMPTIVE_OS_PIPEOPS'
    ,N'PREEMPTIVE_OS_GENERICOPS',N'PREEMPTIVE_OS_VERIFYTRUST'
    ,N'PREEMPTIVE_OS_FILEOPS',N'PREEMPTIVE_OS_DEVICEOPS'
    ,N'PREEMPTIVE_OS_QUERYREGISTRY',N'PREEMPTIVE_XE_CALLBACKEXECUTE'
    ,N'PREEMPTIVE_XE_DISPATCHER',N'PREEMPTIVE_XE_GETTARGETSTATE'
    ,N'PREEMPTIVE_XE_SESSIONCOMMIT',N'PREEMPTIVE_XE_TARGETINIT'
    ,N'PREEMPTIVE_XE_TARGETFINALIZE',N'PWAIT_ALL_COMPONENTS_INITIALIZED'
    ,N'PWAIT_DIRECTLOGCONSUMER_GETNEXT',N'PWAIT_EXTENSIBILITY_CLEANUP_TASK'
    ,N'QDS_PERSIST_TASK_MAIN_LOOP_SLEEP',N'QDS_ASYNC_QUEUE'
    ,N'QDS_CLEANUP_STALE_QUERIES_TASK_MAIN_LOOP_SLEEP'
    ,N'REQUEST_FOR_DEADLOCK_SEARCH',N'RESOURCE_QUEUE',N'SERVER_IDLE_CHECK'
    ,N'SLEEP_BPOOL_FLUSH',N'SLEEP_DBSTARTUP',N'SLEEP_DCOMSTARTUP'
    ,N'SLEEP_MASTERDBREADY',N'SLEEP_MASTERMDREADY',N'SLEEP_MASTERUPGRADED'
    ,N'SLEEP_MSDBSTARTUP',N'SLEEP_SYSTEMTASK',N'SLEEP_TASK'
    ,N'SLEEP_TEMPDBSTARTUP',N'SNI_HTTP_ACCEPT',N'SOS_WORK_DISPATCHER'
    ,N'SP_SERVER_DIAGNOSTICS_SLEEP',N'SQLTRACE_BUFFER_FLUSH'
    ,N'SQLTRACE_INCREMENTAL_FLUSH_SLEEP',N'SQLTRACE_WAIT_ENTRIES'
    ,N'STARTUP_DEPENDENCY_MANAGER',N'WAIT_FOR_RESULTS'
    ,N'WAITFOR',N'WAITFOR_TASKSHUTDOWN',N'WAIT_XTP_HOST_WAIT'
    ,N'WAIT_XTP_OFFLINE_CKPT_NEW_LOG',N'WAIT_XTP_CKPT_CLOSE',N'WAIT_XTP_RECOVERY'
    ,N'XE_BUFFERMGR_ALLPROCESSED_EVENT',N'XE_DISPATCHER_JOIN',N'XE_DISPATCHER_WAIT'
    ,N'XE_LIVE_TARGET_TVF',N'XE_TIMER_EVENT')
)
SELECT
  w1.wait_type AS [Wait Type]
  ,w1.waiting_tasks_count AS [Wait Count]
  ,CONVERT(DECIMAL(12,3), w1.wait_time_ms / 1000.0) AS [Wait Time]
  ,CONVERT(DECIMAL(12,1), w1.wait_time_ms / w1.waiting_tasks_count)
        AS [Avg Wait Time]
  ,CONVERT(DECIMAL(12,3), w1.signal_wait_time_ms / 1000.0)
        AS [Signal Wait Time]
  ,CONVERT(DECIMAL(12,1), w1.signal_wait_time_ms / w1.waiting_tasks_count)
        AS [Avg Signal Wait Time]
  ,CONVERT(DECIMAL(12,3), w1.resource_wait_time_ms / 1000.0)
        AS [Resource Wait Time]
  ,CONVERT(DECIMAL(12,1), w1.resource_wait_time_ms / w1.waiting_tasks_count)
        AS [Avg Resource Wait Time]
  ,CONVERT(DECIMAL(6,3), w1.Pct)
        AS [Percent]
  ,CONVERT(DECIMAL(6,3), w1.RunningPct)
```

```
        AS [Running Percent]
FROM
    Waits w1
WHERE
    w1.RunningPct <= 99 OR w1.RowNum = 1
ORDER BY
    w1.RunningPct
OPTION (RECOMPILE, MAXDOP 1);
```

На рис. 2.3 показан вывод этого кода, запущенного на одном из промышленных серверов на начальных этапах устранения неполадок. Сразу видно, что большинство ожиданий в системе связаны с блокировками (`LCK*`) и вводом/выводом (`PAGEIOLATCH*`). Теперь гораздо легче понять, с чего начать устранение неполадок.

	Wait Type	Wait Count	Wait Time	Avg Wait Time	Signal Wait Time	Avg Signal Wait Time	Resource Wait Time	Avg Resource Wait Time	Percent	Running Percent
1	LCK_M_U	538312358	2952904.553	5.0	278904.170	1.0	2674000.376	4.0	58.316	58.316
2	PAGEIOLATCH_SH	132056495	730022.059	5.0	17938.737	0.0	712083.322	5.0	14.417	72.733
3	LCK_M_S	196405075	379378.938	1.0	24706.314	0.0	354672.624	1.0	7.492	80.226
4	ASYNC_NETWORK_IO	36665258	254793.758	6.0	100063.339	2.0	154730.419	4.0	5.032	85.258
5	LOGBUFFER	11718571	165042.270	14.0	18931.562	1.0	146110.708	12.0	3.259	88.517
6	PAGEIOLATCH_EX	153474407	133057.566	0.0	3225.941	0.0	129831.625	0.0	2.628	91.145
7	LCK_M_IX	496185	98525.504	198.0	139.082	0.0	98386.422	198.0	1.946	93.091
8	IO_COMPLETION	93217317	81833.420	0.0	3505.294	0.0	78328.126	0.0	1.616	94.707
9	LATCH_EX	49863173	65876.146	1.0	10921.396	0.0	54954.750	1.0	1.301	96.008
10	ASYNC_IO_COMPLETION	57845	56036.933	968.0	22.078	0.0	56014.855	968.0	1.107	97.114
11	LCK_M_IS	57448	31694.644	551.0	9.403	0.0	31685.241	551.0	0.626	97.740
12	LCK_M_SCH_M	2228	31016.126	13921.0	0.918	0.0	31015.208	13920.0	0.613	98.353
13	WRITELOG	1969821	26014.687	13.0	715.277	0.0	25299.410	12.0	0.514	98.867
14	OLEDB	3041936	14911.992	4.0	6058.799	1.0	8853.193	2.0	0.294	99.161

Рис. 2.3. Пример вывода `sys.dm_os_wait_stats`

Этот подход называется *анализом статистики ожидания*. Это один из самых популярных методов устранения неполадок и настройки производительности в SQL Server. На рис. 2.4 показан типичный цикл устранения неполадок, когда применяется анализ статистики ожидания.

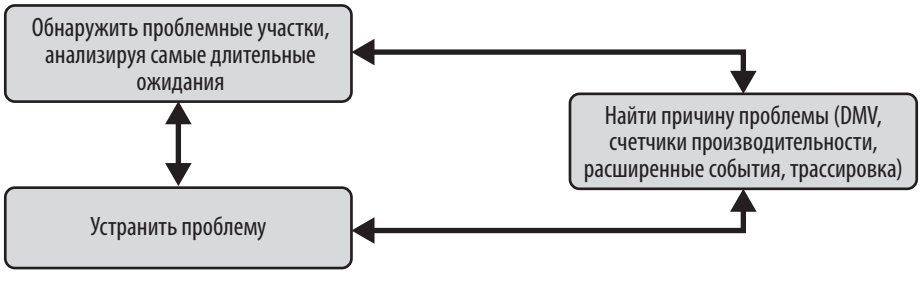


Рис. 2.4. Типичный цикл устранения неполадок с помощью анализа статистики ожидания

Сначала вы выявляете главное узкое место, анализируя самые длительные ожидания. Затем с помощью других инструментов и методов вы убеждаетесь, что это действительно главное узкое место, и отыскиваете основную причину проблемы. Наконец, вы устраняете проблему и повторяете цикл.

Этот процесс рискует никогда не закончиться, потому что всегда есть что улучшить. Но с какого-то момента дальнейшие улучшения уже не окупают усилий. Помните принцип Парето: на 80 % улучшений вы потратите 20 % своего времени, так что не расходуйте его на малосущественные настройки.

Теоретически все выглядит очень просто, но, к сожалению, жизнь устроена сложнее. Многие проблемы тесно связаны друг с другом, отчего становится труднее обнаружить настоящие причины узких мест. Вот распространенный пример: чрезвычайно долгие ожидания диска часто вызваны не плохой производительностью ввода/вывода, а плохо оптимизированными запросами, которые постоянно очищают буферный пул и перегружают дисковую подсистему.

На рис. 2.5 показаны примеры высокоуровневых зависимостей, с которыми вы можете столкнуться. Эта схема не охватывает всех возможных ситуаций, но иллюстрирует, как важен широкий взгляд на систему при устранении неполадок.

Я думал составить список наиболее распространенных ожиданий и их возможных причин, однако не хочу, чтобы вы лечили симптомы вместо болезни. Вместо того чтобы полагаться на «волшебный» список, лучше прочтите всю книгу, чтобы разобраться в возможных зависимостях.

Конкретные проблемы и методы устранения неполадок мы разберем в следующих главах, а пока давайте рассмотрим важные динамические административные представления в SQL Server, связанные с SQLOS и моделью выполнения SQL Server.

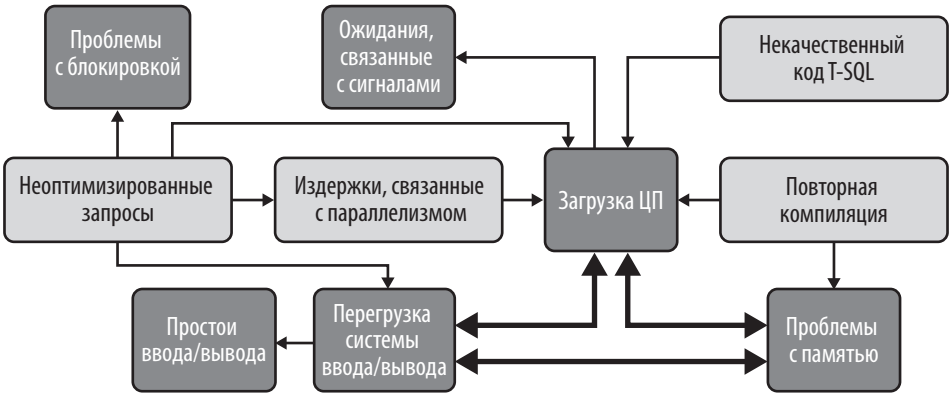


Рис. 2.5. Зависимости и проблемы

Динамические административные представления, связанные с моделью выполнения

В SQL Server много динамических административных представлений. Чтобы узнать о них подробнее, обратитесь к документации Microsoft[1]. В этом разделе я рассмотрю только избранные представления, которыми регулярно пользуюсь при устранении неполадок. Дальше в книге встретятся и многие другие представления.

sys.dm_os_wait_stats

Как мы уже видели, представление `sys.dm_os_wait_stats`[2] информирует об ожиданиях в системе. Оно сообщает, сколько раз инициировалось ожидание (`waiting_task_count`), а также совокупную продолжительность ожиданий ресурсов (`resource_wait_time_ms`) и ожиданий сигналов (`signal_wait_time_ms`). Время ожидания ресурса — это время, в течение которого задача находилась в очереди `SUSPENDED`, ожидая ресурса. Время ожидания сигнала — это время, в течение которого ЦП находился в очереди `RUNNABLE` после того, как время ожидания ресурса закончилось.

Для примера рассмотрим задачу, которая должна прочитать страницу данных с диска. Запрос ввода/вывода может занять 6 мс, затем задача может подождать еще одну миллисекунду, прежде чем возобновить выполнение. Если вы просмотрите статистику ожидания, то увидите 6 мс ожидания ресурса, 1 мс ожидания сигнала и 7 мс общего времени ожидания.

---

[1] https://oreil.ly/TqA24

[2] https://oreil.ly/offh8

В листинге 2.2 показано, как можно сравнить совокупное время ожидания сигналов и ожидания ресурсов.

Листинг 2.2. Сравнение времени ожидания сигналов и ресурсов

```
SELECT
    SUM(signal_wait_time_ms) AS [Signal Wait Time (ms)  ]
    ,CONVERT(DECIMAL(7,4), 100.0 * SUM (signal_wait_time_ms) /
        SUM(wait_time_ms)) AS [% Signal waits]
    ,SUM(wait_time_ms - signal_wait_time_ms) AS [Resource Wait Time (ms)]
    ,CONVERT (DECIMAL(7,4), 100.0 * sum(wait_time_ms - signal_wait_time_ms) /
        SUM(wait_time_ms)) AS [% Resource waits]
FROM
    sys.dm_os_wait_stats WITH (NOLOCK);
```

Как правило, ожидание сигнала (signal wait time) не должно занимать больше 10–15 % от общего времени ожидания (wait time). Большее значение может указывать на узкое место ЦП, когда задачи проводят много времени в очереди `RUNNABLE`. Однако не спешите делать вывод, что нужно просто добавить процессоров: может оказаться, что проблема полностью решается с помощью настройки производительности.

Обратите внимание на то, как часто происходят ожидания. Иногда возникают ожидания с низким значением `wait_task_count` и больши́м общим временем ожидания. Возможно, вам и не потребуется анализировать соответствующие задачи, особенно на начальном этапе устранения неполадок. Такие ожидания часто вызываются производственными инцидентами или другими нетипичными условиями.

Наконец, убедитесь, что вы анализируете репрезентативные данные. Как я уже говорил, статистика собирается с момента последнего перезапуска SQL Server, а рабочая нагрузка на сервер может меняться со временем.

Обычно я прошу клиентов очистить информацию об ожиданиях, оставив данные только за несколько дней до начала устранения неполадок. Команду `DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR)` можно безопасно использовать на промышленном сервере, хотя она может повлиять на сбор данных в некоторых сторонних инструментах мониторинга.

Как вариант, можно получить статистику ожидания дважды и анализировать разницу между значениями. Сценарий для этого приведен в сопутствующих материалах этой книги.

sys.dm_exec_session_wait_stats

Начиная с SQL Server 2016, информацию об ожиданиях можно просматривать на уровне сеанса с помощью представления `sys.dm_exec_session_wait_stats`[1].

[1] https://oreil.ly/iZ9uJ

Оно чрезвычайно полезно, когда нужно бороться с долго выполняющимися запросами или медленными хранимыми процедурами. В этом представлении вы увидите ожидания, которые произошли во время выполнения, что поможет точнее определить узкие места и участки, которые стоит изучить.

Столбцы и данные в этом представлении почти такие же, как в `sys.dm_os_wait_stats`. Можно легко модифицировать сценарии, чтобы они работали в обоих случаях. Имейте в виду, что данные в `sys.dm_exec_session_wait_stats` очищаются при открытии сеанса и при сбросе соединения из пула.

Возможно, вы обратите внимание, что в представлении не всегда обновляются сведения об инструкциях, запущенных в данный момент. Чтобы увидеть эти сведения, нужно дождаться, пока закончится обработка запроса.

sys.dm_os_waiting_tasks

В представлении `sys.dm_os_waiting_tasks`[1] отображается список задач, которые *прямо сейчас* ожидают в очереди `SUSPENDED`. Это представление удобно, когда сервер перегружен или не отвечает и вы хотите понять, почему сеансы приостановлены. Оно также полезно при устранении проблем с конкурентным доступом и активными блокировками в системе, потому что показывает идентификатор сеанса блокировщика для той или иной задачи (подробнее об этом в главе 8).

Наиболее полезные столбцы в этом представлении:

`session_id`

Идентификатор ожидающего сеанса.

`wait_type`

Тип ожидания, которое задерживает сеанс.

`wait_duration_ms`

Продолжительность ожидания.

`blocking_session_id`

Сеанс, блокирующий текущую задачу. Как я уже упоминал, этот столбец чрезвычайно полезен при анализе активных блокировок в системе.

`resource_address`

Информация о ресурсе, доступности которого ожидает задача.

Если вы имеете дело с параллельными планами выполнения, представление `sys.dm_os_waiting_tasks` может выводить более одной строки на сеанс.

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

sys.dm_exec_requests

Представление `sys.dm_exec_requests`[1] дает подробную информацию о каждом запросе, который в данный момент выполняется на сервере. Это удобный мгновенный снимок того, что происходит прямо сейчас, и с его помощью можно выявить, какие из выполняющихся сейчас запросов особенно интенсивно используют ЦП или ввод/вывод.

Это представление возвращает информацию как для пользовательских, так и для системных сеансов. Чтоб отфильтровать большинство системных сеансов, можно использовать предикат `WHERE session_id > 50`, хотя в современных конфигурациях бывают и системные сеансы с идентификаторами, превышающими 50. Можно также отфильтровать системные процессы: для этого объедините данные с представлением `sys.dm_exec_sessions` и используйте там столбец `is_user_process`.

Наиболее полезные столбцы в этом представлении:

`session_id`

Идентификатор сеанса. В отличие от `sys.dm_os_waiting_tasks`, каждому сеансу соответствует одна строка, если только вы не используете несколько активных результирующих наборов (MARS, Multiple Active Result Set).

`start_time`

Время начала запроса.

`total_elapsed_time`

Продолжительность запроса.

`status`

Текущий статус запроса (`RUNNING`, `RUNNABLE`, `SUSPENDED`, `SLEEPING`). Состояние `SLEEPING` означает, что соединение простаивает.

`wait_type`, `wait_time`, `wait_resource`, `blocking_session_id`

Выводятся, если запрос в данный момент приостановлен. Как и в случае с `sys.dm_os_waiting_tasks`, столбец `blocking_session_id` полезен при устранении неполадок активной блокировки в системе.

`cpu_time`, `logical_reads`, `reads`, `writes`, `granted_query_memory`, `dop`

Выводят метрики выполнения.

`sql_handle`, `plan_handle`

Позволяют получить инструкцию и план ее выполнения.

---

[1] https://oreil.ly/wJJe2

В листинге 2.3 приведен код, который возвращает информацию о текущих запросах, интенсивно использующих ЦП, а также данные о соединении. Этот код работает в версиях SQL Server 2016 и новее. Для более старых версий можно удалить столбец `ep.dop`. Кроме того, для версий ниже, чем SQL Server 2012, удалите функцию `TRY_CONVERT`.

Листинг 2.3. Использование представления `sys.dm_exec_requests`

```
SELECT
    er.session_id
    ,er.request_id
    ,DB_NAME(er.database_id) as [database]
    ,er.start_time
    ,CONVERT(DECIMAL(21,3),er.total_elapsed_time / 1000.) AS [duration]
    ,er.cpu_time
    ,SUBSTRING(
        qt.text,
        (er.statement_start_offset / 2) + 1,
            ((CASE er.statement_end_offset
                WHEN -1 THEN DATALENGTH(qt.text)
                ELSE er.statement_end_offset
            END – er.statement_start_offset) / 2) + 1
    ) AS [statement]
    ,er.status
    ,er.wait_type
    ,er.wait_time
    ,er.wait_resource
    ,er.blocking_session_id
    ,er.last_wait_type
    ,er.reads
    ,er.logical_reads
    ,er.writes
    ,er.granted_query_memory
    ,er.dop
    ,er.row_count
    ,er.percent_complete
    ,es.login_time
    ,es.original_login_name
    ,es.host_name
    ,es.program_name
    ,c.client_net_address
    ,ib.event_info AS [buffer]
    ,qt.text AS [sql]
    ,TRY_CONVERT(XML,p.query_plan) as [query_plan]
FROM
    sys.dm_exec_requests er WITH (NOLOCK)
        OUTER APPLY sys.dm_exec_input_buffer
            (er.session_id, er.request_id) ib
        OUTER APPLY sys.dm_exec_sql_text(er.sql_handle) qt
        OUTER APPLY
            sys.dm_exec_text_query_plan
            (
                er.plan_handle
```

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

Получать план выполнения запроса с помощью функции `sys.dm_exec_text_query_plan` — весьма ресурсоемко. Закомментируйте ее, если ваш сервер работает с большой нагрузкой на процессор.

sys.dm_os_schedulers

Представлением `sys.dm_os_schedulers`[1] я пользуюсь лишь время от времени. Как можно догадаться по названию, оно содержит информацию о планировщиках в системе. Его можно применять, чтобы получить сведения о распределении планировщиков по узлам NUMA, а также чтобы анализировать показатели отдельных планировщиков.

В главе 1 я привел код для первого варианта использования, а теперь рассмотрим расширенную версию этого сценария (см. листинг 2.4). Проверьте количество активных планировщиков на каждом узле NUMA, чтобы убедиться, что соответствие ЦП установлено правильно.

Листинг 2.4. Статистика планировщика узла NUMA

```
SELECT
    parent_node_id AS [NUMA Node]
    ,COUNT(*) AS [Schedulers]
    ,SUM(IIF(status = N'VISIBLE ONLINE',1,0)) AS [Online Schedulers]
    ,SUM(IIF(status = N'VISIBLE OFFLINE',1,0)) AS [Offline Schedulers]
    ,SUM(current_tasks_count) AS [Current Tasks]
    ,SUM(runnable_tasks_count) AS [Runnable Tasks]

FROM
    sys.dm_os_schedulers WITH (NOLOCK)
WHERE
    status IN (N'VISIBLE ONLINE',N'VISIBLE OFFLINE')
GROUP BY
    parent_node_id
OPTION (RECOMPILE, MAXDOP 1);
```

[1] https://oreil.ly/8CfB0

В столбцах `current_tasks_count` и `runnable_tasks_count` указано количество задач в очередях `RUNNING` и `RUNNABLE` на каждом узле. Большое число `runnable_tasks_count` может говорить об узком месте на уровне процессора. Но не забывайте, что эти числа отражают состояние системы *в данный конкретный момент* и могут стать неактуальными для следующего момента. Более полную картину даст сводная информация: например, процентная доля ожидания сигнала (см. листинг 2.2) или время загрузки ЦП (см. главу 6).

В других столбцах в этом представлении приведены показатели, специфические для планировщика, например статус, количество исполнителей и задач в разных состояниях, количество переключений контекста, уровень потребления ЦП и некоторые другие. Более подробные сведения можно найти в документации.

Обзор регулятора ресурсов

Регулятор ресурсов (Resource Governor) — это инструмент Enterprise Edition, позволяющий разделять и лимитировать различные рабочие нагрузки на сервере. Он появился довольно давно, но для меня так и остался нишевой функцией: я редко сталкиваюсь с ним в реальной жизни. (Можно даже пропустить этот раздел и вернуться к нему позже, если понадобится.) Однако не забудьте проверить, настроен ли регулятор ресурсов в системе, с которой вы работаете, потому что его неправильная конфигурация может серьезно ухудшить быстродействие сервера.

Когда регулятор ресурсов включен, он разделяет сеансы между различными *группами рабочей нагрузки* путем вызова *функции классификатора* во время входа в сеанс. Функция классификатора — это простая пользовательская функция, которая позволяет выбирать группу рабочей нагрузки по различным свойствам соединения (идентификатор учетной записи, имя приложения, IP-адрес клиента и т. д.).

У каждой группы рабочей нагрузки есть несколько параметров, например `MAXDOP`, максимально допустимое процессорное время на выполнение запроса и максимально допустимое количество одновременных запросов в группе. Группы также связаны с пулом ресурсов, который позволяет регулировать потребление ресурсов.

В документации по SQL Server *пулы ресурсов* описаны как «виртуальные экземпляры SQL Server внутри экземпляра SQL Server». Мне это определение кажется неточным, потому что пулы ресурсов недостаточно изолированы друг от друга. Тем не менее с ними можно контролировать и ограничивать пропускную способность ЦП и соответствие планировщиков, а также выделение памяти для запросов (см. главу 7).

Начиная с SQL Server 2014, можно также регулировать пропускную способность диска, ограничивая для пула ресурсов количество операций ввода/вывода в единицу времени (IOPS). Правда, нельзя контролировать использование буферного пула: он является общим для всех пулов.

Существуют две системные группы рабочих нагрузок: *внутренняя группа* и *группа по умолчанию*. С каждой из них связан собственный пул ресурсов. Как вы можете догадаться, первая группа обрабатывает внутреннюю рабочую нагрузку, а вторая отвечает за всю неклассифицированную нагрузку. Можно изменять параметры группы рабочей нагрузки *по умолчанию*, не создавая лишних пользовательских групп рабочей нагрузки и соответствующих пулов.

На рис. 2.6 показана конфигурация регулятора ресурсов в случае, когда вы хотите разделить рабочие нагрузки OLTP и отчетов. В результате запросы отчетов будут меньше влиять на критические транзакции OLTP, которые грозили выжечь ваш ЦП и устройства ввода/вывода.

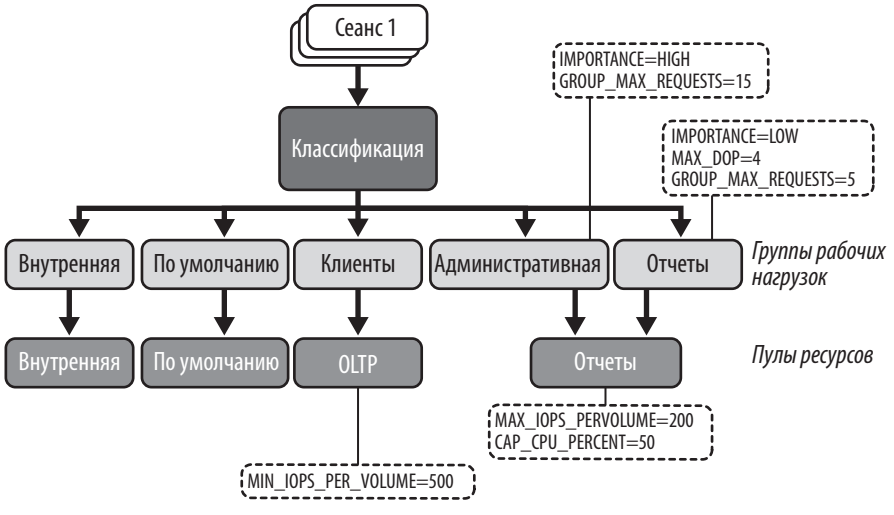


Рис. 2.6. Пример конфигурации регулятора ресурсов

Регулятор ресурсов — полезная штука, но настраивать и обслуживать его непросто. Если вы захотите лимитировать использование ресурсов для нескольких высоконагруженных пулов ресурсов, это потребует специального планирования и расчетов.

Также нужно время от времени перепроверять настройки, потому что требования к оборудованию и рабочей нагрузке могут меняться. Недавно мне пришлось устранять неполадки в ситуации, когда обновление главной дисковой подсистемы не улучшило производительность. Мы обнаружили, что скорость ввода/вывода в системе была вручную лимитирована параметром `MAX_IOPS_PER_VOLUME` в пуле ресурсов.

В заключение замечу, что регулятор ресурсов удобен, когда нужно разделить различные рабочие нагрузки в одной и той же базе данных на отдельном сервере или в экземпляре, где используется отказоустойчивая кластеризация. Регулятор ресурсов также помогает уменьшить влияние операций по обслуживанию

базы данных. Например, с его помощью можно ограничить расход ресурсов ЦП на сжатие резервных копий или лимитировать нагрузку ввода/вывода от обслуживания индексов, запуская эти задачи в пуле ресурсов, отдельном от пользовательских сеансов.

Рекомендую обратить внимание на альтернативные технологии, когда нужно разделить рабочую нагрузку в группах доступности Always On. Вторичные реплики могут обеспечить лучшую масштабируемость в долгосрочной перспективе. Кроме того, если требуется разделить рабочие нагрузки нескольких баз данных, работающих на одном экземпляре SQL Server, обычно лучшим решением будет распределить базы данных по нескольким экземплярам и, возможно, виртуализировать их.

Резюме

SQLOS — жизненно важная подсистема, отвечающая за планирование и управление ресурсами в SQL Server. Во время запуска она создает планировщики — по одному на каждый логический ЦП — и выделяет каждому планировщику пул рабочих потоков. Пользовательские и системные задачи назначаются рабочим потокам, которые выполняют всю работу.

В SQL Server используется совместное планирование, когда исполнители добровольно уступают управление через каждые 4 мс. Задачи постоянно перемещаются между состояниями `RUNNING` (выполняется на ЦП), `SUSPENDED` (ожидает ЦП) и `RUNNABLE` (ожидает ресурсов). SQL Server отслеживает различные типы ожидания и выводит статистику по ним в представлении `sys.dm_os_wait_tasks`. Анализировать наиболее выделяющиеся типы ожидания и выявлять узкие места в системе можно с помощью методики, которая называется статистикой ожидания.

Анализируя ожидания, проявляйте осторожность и не делайте поспешных выводов. Проблемы производительности часто связаны друг с другом и маскируют друг друга. В ходе анализа важно выявить и подтвердить основную причину проблемы.

В следующей главе мы углубимся в поиск и устранение конкретных неполадок, начиная с проблем дисковой подсистемы.

Чек-лист устранения неполадок

- ☐ Пронаблюдать ожидания в системе. Убедиться, что статистика ожидания репрезентативна.
- ☐ Проанализировать процентные доли ожидания сигналов и ресурсов.
- ☐ Проверить конфигурацию регулятора ресурсов, если он используется.
- ☐ Отсортировать типы ожиданий по продолжительности, чтобы было легче найти узкие места.

ГЛАВА 3

Производительность дисковой подсистемы

SQL Server очень интенсивно занимается вводом/выводом: данные постоянно считываются с диска и записываются на него. Для производительности и исправной работы SQL Server важна хорошая пропускная способность ввода/вывода. К сожалению, во многих экземплярах SQL Server, даже с современными хранилищами на основе флеш-памяти, быстродействие ограничено именно вводом/выводом.

В этой главе я покажу, как анализировать и устранять проблемы с производительностью дисковой подсистемы. Вы познакомитесь с внутренними механизмами обработки запросов в SQL Server и научитесь обнаруживать возможные узкие места во всем стеке ввода/вывода: на уровне SQL Server, ОС, виртуализации и хранилища.

Затем пойдет речь о настройке контрольных точек, с которыми связаны типичные узкие места ввода/вывода в высоконагруженных системах OLTP.

Наконец, я перечислю наиболее распространенные типы ожидания, связанные с вводом/выводом.

Как устроена подсистема ввода/вывода SQL Server

SQL Server никогда не работает со страницами данных непосредственно в файлах базы данных. Каждый раз, когда нужно прочитать или изменить страницу данных, SQL Server считывает ее в память и кэширует в *буферном пуле*. На каждую страницу в буферном пуле ссылается *буферная структура*, иногда называемая просто *буфером*. В ней содержатся адрес страницы в файле данных, указатель на страницу данных в памяти, информация о состоянии и очередь кратковременных блокировок страниц.

В SQL Server используются *кратковременные блокировки (latches)*, которые защищают внутренние объекты в памяти от повреждения, если несколько потоков пытаются изменять их одновременно. Два самых распространенных типа кратковременных блокировок — *монопольные (EX latch)*, которые блокируют любой доступ к объекту, и *совмещаемые (SH latch)*, которые разрешают одновременное чтение, но не дают изменять объекты.

Концептуально кратковременные блокировки похожи на *критические секции* или *мьютексы* в программировании. Подробнее о кратковременных блокировках я расскажу в главе 10.

В общем случае страницы данных в буферном поле размещаются не в том порядке, в котором они хранятся в файлах базы данных. Несмотря на это, SQL Server эффективно находит страницы данных в буферном пуле. Когда SQL Server обращается к странице в пуле, он выполняет *логическое чтение*. Если страницы нет в памяти и ее нужно прочитать с диска, происходит также *физическое чтение*.

Когда нужно изменить данные, SQL Server создает записи журнала и заносит их в файл журнала транзакций, а затем модифицирует страницы в буферном пуле, помечая их как «*грязные*». Сервер асинхронно сохраняет «грязные» страницы в файлы данных в *контрольных точках*, а иногда в процессах *отложенной записи*. Мы обсудим оба этих процесса позже в этой главе, а журналы транзакций — в главе 11. Имейте в виду: чтобы модифицировать данные, SQL Server вынужден *считывать* страницы данных с диска, если только они не были кэшированы.

Теперь подробнее поговорим о том, как SQL Server работает с вводом/выводом.

Планирование и ввод/вывод

В главе 2 мы говорили, что в SQL Server используется совместное планирование, и на процессорах поочередно работает по несколько исполнителей, или рабочих процессов (workers). Они добровольно уступают управление по истечении 4 мс, позволяя другим исполнителям продолжить работу. Эта схема требует, чтобы ввод/вывод в SQL Server происходил максимально асинхронно: если бы исполнители ждали результатов запросов ввода/вывода, они не давали бы работать другим исполнителям.

По умолчанию все планировщики SQL Server обрабатывают запросы ввода/вывода. Это поведение можно перенастроить: привязать ввод/вывод к определенным ЦП, установив маску соответствия ввода/вывода. Теоретически это может улучшить пропускную способность ввода/вывода в высоконагруженных системах OLTP, однако чаще всего мне эта мера не кажется нужной. Обычно гораздо лучшие результаты приносят оптимизация и снижение нагрузки на ЦП и ввод/вывод.

О масках соответствия можно почитать в документации Microsoft[1].

У каждого планировщика есть своя выделенная очередь ввода/вывода. Когда исполнителю надо выполнить операцию ввода/вывода, он создает *структуру запроса* ввода/вывода, добавляет ее в очередь планировщика и выполняет асинхронный вызов ввода/вывода через API операционной системы. После этого исполнитель уже не ждет, пока запрос завершится: он продолжает работать, занимаясь другими делами, или приостанавливает работу и перемещается в очередь `SUSPENDED`.

Когда в планировщике начинает выполняться новый рабочий процесс (переключаясь в состояние `RUNNING`), он проходит через очередь ввода/вывода планировщика. Структуры запроса ввода/вывода содержат всю необходимую информацию, чтобы проверить, завершился ли асинхронный вызов API ОС, и хранят указатель на функцию обратного вызова, которую исполнитель вызывает по завершении запроса.

Ну да, звучит сложно. Пожалуйста, еще немного терпения: мы рассмотрим подробности в следующем разделе. Вот ключевые моменты, которые я советую запомнить:

- Все активные планировщики по умолчанию обрабатывают запросы ввода/вывода.
- Большинство запросов ввода/вывода в SQL Server работают через асинхронные вызовы API ОС. Это касается даже протоколирования с опережающей записью: исполнитель, который инициировал инструкцию `COMMIT`, может быть приостановлен, пока запись журнала не запишется на диск; однако команда записи API ОС выполнится асинхронно.
- Запрос ввода/вывода может выполнять не тот исполнитель, который вызвал этот запрос.

Список ожидающих запросов ввода/вывода можно найти в представлении `sys.dm_io_pending_io_requests`. В столбце `io_pending_ms_ticks` показана продолжительность запроса. Столбец `io_pending` указывает, завершился ли вызов API ОС и ожидает ли запрос, чтобы исполнитель его закончил. Это может помочь разобраться, влияет ли загрузка ЦП на задержку запросов.

А теперь, как я и обещал, давайте еще раз рассмотрим процедуру ввода/вывода на более конкретных примерах чтения страниц данных с диска.

---

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

Чтение данных

Когда SQL Server нужна страница данных, он проверяет, есть ли она уже в буферном пуле. Если нет, исполнитель выделяет для страницы буфер, защищая ее монопольной кратковременной блокировкой. Это не позволяет рабочим процессам обращаться к странице, пока она не будет прочитана.

После этого исполнитель создает структуру запроса ввода/вывода, помещает ее в очередь ввода/вывода планировщика и инициирует запрос чтения через API ОС. Затем он пытается получить еще одну общую кратковременную блокировку в буфере, но буфер уже заблокирован несовместимой монопольной блокировкой. Поэтому рабочий процесс приостанавливается, переходя в состояние ожидания `PAGEIOLATCH` (это показано на рис. 3.1).

Когда другой исполнитель переходит в состояние `RUNNING`, он проверяет, нет ли в очереди планировщика выполненных запросов ввода/вывода. Если есть, то исполнитель вызывает функцию обратного вызова для завершения операции: функция проверяет, что страница не повреждена, и удаляет монопольную кратковременную блокировку из буфера. Исполнитель, который отправил запрос ввода/вывода, может возобновить работу и получить доступ к странице данных (рис. 3.2).

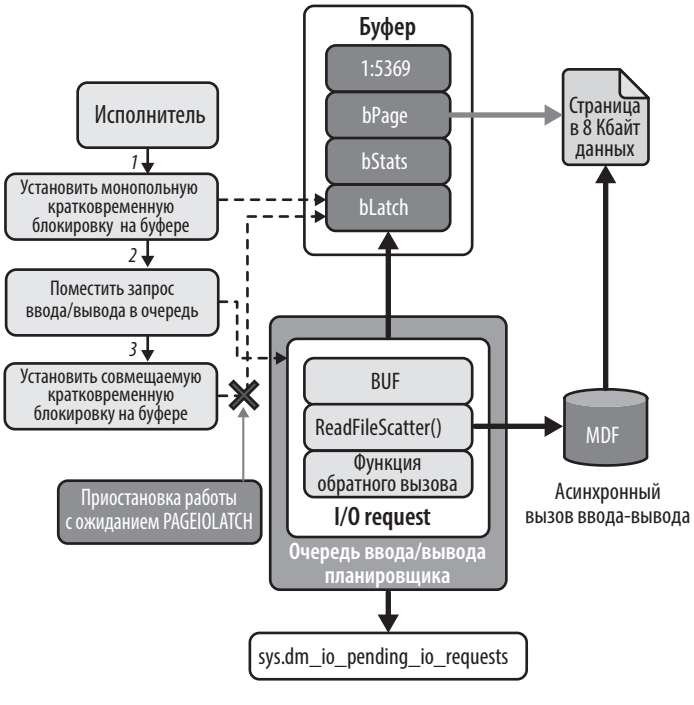


Рис. 3.1. Как инициируется чтение страницы данных с диска

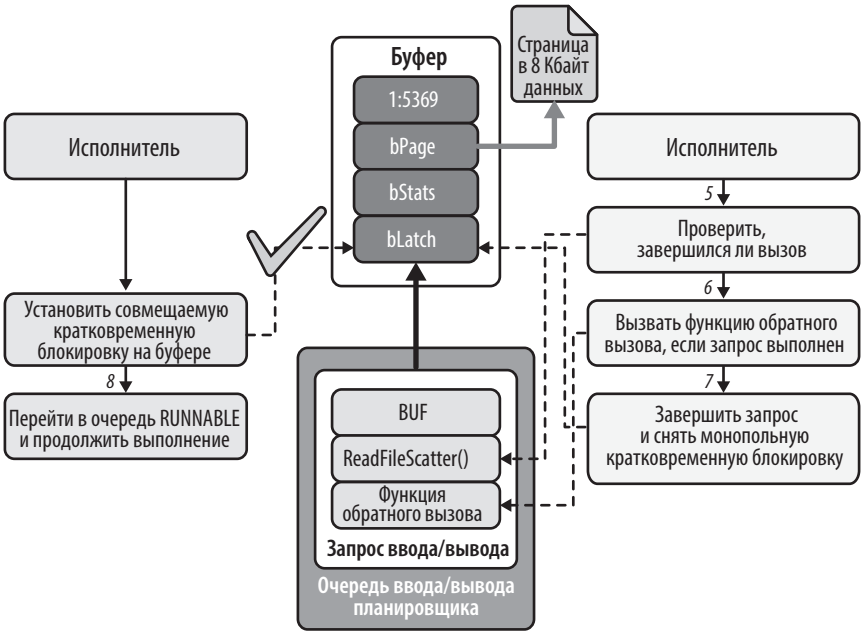


Рис. 3.2. Как завершается чтение страницы данных с диска

Во время запросов ввода/вывода могут возникнуть определенные ошибки. Все они серьезные, и для них нужно настроить уведомления.

- Ошибка 823 означает, что вызов API ввода/вывода ОС завершился неудачно. Часто это признак аппаратных проблем.
- Ошибки 605 и 824 указывают на проблемы с логической согласованностью страниц данных. Если вы столкнулись с одной из этих ошибок, немедленно проверьте с помощью команды `DBCC CHECKDB`, не повреждена ли база данных. Эти ошибки также бывают из-за неисправных драйверов ввода/вывода, которые могут повредить страницы данных во время передачи.
- Ошибка 833 означает, что возврат из запроса ввода/вывода (вызов API ОС) занял больше 15 секунд. Это ненормально: в случае такой ошибки проверьте, исправна ли дисковая подсистема.
- Ошибка 825 означает, что запрос ввода/вывода не сработал, и его надо повторить. Как и в случае с ошибкой 833, проверьте дисковую подсистему.

Устраняя эти ошибки, подробности можно уточнять в журнале ошибок SQL Server (с помощью кода из листинга 1.4) и в журнале системных событий.

SQL Server часто читает несколько страниц данных одним запросом ввода/вывода. Например, он использует логику упреждающего чтения, считы-

вая несколько страниц данных во время просмотра. В результате запрос может выполнять тысячи логических операций чтения всего за несколько физических операций. Еще один пример — *чтение с ускорением (ramp-up reads)*, когда во время запуска SQL Server считывает большое количество страниц при каждом запросе ввода/вывода, пытаясь быстро заполнить буферный пул.

Запись данных

SQL Server обрабатывает запись данных примерно так же, как чтение. В большинстве случаев запись выполняется асинхронно с помощью очередей ввода/вывода планировщика, как было показано в предыдущих примерах. Очевидно, что в разных операциях ввода/вывода функция обратного вызова реализуется по-разному.

Когда вы модифицируете данные в базе, SQL Server модифицирует страницы данных в буферном пуле, при необходимости читая страницы с диска. Он регистрирует изменения и заносит соответствующие записи в журнал транзакций. Транзакция не считается принятой, пока записи журнала не сохранятся на диске. Хотя *формально* можно рассматривать опережающее протоколирование как синхронную запись, в SQL Server для записи в журнал используется асинхронный ввод/вывод.

SQL Server асинхронно записывает измененные страницы данных с помощью *процесса контрольных точек (check-point process)*: он находит «грязные» страницы данных в буферном пуле и сохраняет их на диск. Он пытается свести к минимуму количество запросов к диску, по возможности объединяя смежные измененные страницы и записывая их за одну операцию ввода/вывода.

Существует и другой процесс SQL Server, называемый *отложенной записью*, который периодически очищает буферный пул: он удаляет страницы данных, к которым в последнее время не обращались, и тем самым высвобождает память. В нормальных условиях процесс отложенной записи не обрабатывает «грязные» страницы данных, однако иногда может тоже записывать их на диск, если в системе не хватает памяти.

Как всегда, бывают исключения. Например, во время операции массового импорта SQL Server выделяет набор буферов в буферном пуле и использует их повторно, записывая данные в базу вне контрольной точки. Он сохраняет содержимое буферного пула, чтобы тот не очищался при импорте больших объемов данных.

Ввод/вывод в контрольных точках может вызывать проблемы в высоконагруженных системах. Я расскажу о настройке контрольных точек позже в этой главе, но сначала давайте целостно рассмотрим всю подсистему хранения.

Подсистема хранения: целостный обзор

Бороться с медленным вводом/выводом в SQL Server — непростая задача. Я видел немало жарких споров между специалистами по базам данных и по инфраструктуре. Специалисты по базам данных обычно жалуются на низкую производительность диска, а специалисты по устройствам хранения анализируют микроскопические показатели задержки от устройств SAN и настаивают, что во всем виноват SQL Server. На самом деле и те и другие неправы. Обычно они совершают одну и ту же ошибку: рассматривают систему хранения чересчур упрощенно, будто она состоит всего из пары компонентов. Это не так.

На рис. 3.3 изображена схема сетевой подсистемы хранения данных, показанная на самом верхнем уровне, без особых подробностей. (На схеме также упоминаются отдельные средства устранения неполадок, которые мы рассмотрим позже. На них пока можно не заострять внимания.) Смысл здесь в том, что плохую производительность ввода/вывода может вызвать любой компонент, поэтому нужно проанализировать *все* уровни в стеке.

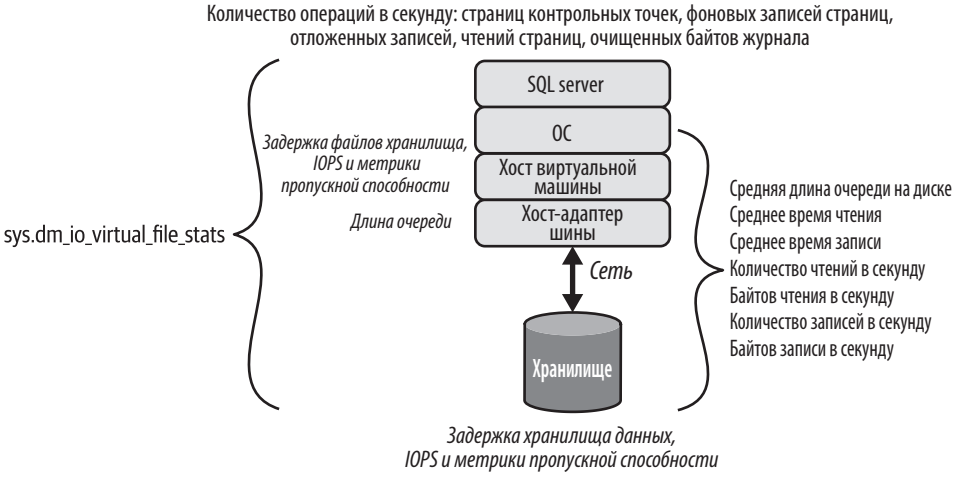


Рис. 3.3. Сетевая подсистема хранения

Также можно использовать хранилище с прямым подключением (DAS, direct-attached storage). В этой конфигурации устройство хранения либо монтируется локально на сервере (советую использовать контроллеры NVMe), либо подключается к нему напрямую. Эта схема исключает сеть из пути к хранилищу и может повысить производительность ввода/вывода. Но есть и недостаток: вы лишаетесь гибкости, которая во внешнем хранилище позволяла бы добавлять дисковое пространство и выполнять обслуживание прозрачно для сервера.

У каждой подсистемы хранения есть переломный момент, после которого задержка запросов ввода/вывода начинает экспоненциально расти с увеличением пропускной способности и IOPS. Например, при рабочей нагрузке 1000 IOPS вы можете получить ответ за 1 мс, а при нагрузке 50 000 IOPS — уже за 3 мс. А затем, когда на уровне 100 000 IOPS наступит переломный момент, значения задержки станут двузначными или даже трехзначными.

У каждого компонента в стеке — свой переломный момент. Например, при малой емкости очереди в хост-адаптере шины могут возникнуть очереди на уровне контроллера, если будет расти количество запросов ввода/вывода. В этом случае SQL Server будет страдать от высоких задержек и плохой производительности ввода/вывода, но при этом никакие метрики SAN не покажут ни малейших задержек.

Чтобы проверять производительность подсистемы хранения, можно использовать утилиту `DiskSpd`, которая эмулирует рабочую нагрузку SQL Server в системе. Утилиту можно скачать с GitHub[1].

Как я уже отмечал, при борьбе с плохой производительностью ввода/вывода нужно проверить все компоненты подсистемы хранения. При этом стоит начать с того, чтобы проанализировать общую задержку хранилища и количество данных, которые SQL Server считывает и записывает в единицу времени. В этом поможет представление `sys.dm_io_virtual_file_stats`.

Представление sys.dm_io_virtual_file_stats

Представление `sys.dm_io_virtual_file_stats`[2] — самый важный инструмент для устранения неполадок с производительностью ввода/вывода. Оно дает статистику ввода/вывода по файлу базы данных, включая количество операций ввода/вывода, объем прочитанных и записанных данных, а также информацию о задержках, связанных с выполнением запросов ввода/вывода.

Данные в этом представлении кумулятивны и рассчитываются с момента перезапуска SQL Server. Сделайте два снимка данных и вычислите разницу между ними (для этого можно использовать код из листинга 3.1). Этот код отфильтровывает файлы базы данных с малой активностью ввода/вывода, потому что их показатели обычно искажены и не очень полезны.

Листинг 3.1. Использование представления `sys.dm_io_virtual_file_stats`

```
DROP TABLE IF EXISTS #Snapshot;
GO
```

[1] https://aka.ms/diskspd

[2] https://oreil.ly/mJCKm

```
CREATE TABLE #Snapshot
(
    database_id SMALLINT NOT NULL,
    file_id SMALLINT NOT NULL,
    num_of_reads BIGINT NOT NULL,
    num_of_bytes_read BIGINT NOT NULL,
    io_stall_read_ms BIGINT NOT NULL,
    num_of_writes BIGINT NOT NULL,
    num_of_bytes_written BIGINT NOT NULL,
    io_stall_write_ms BIGINT NOT NULL
);

INSERT INTO #Snapshot(database_id,file_id,num_of_reads,num_of_bytes_read
    ,io_stall_read_ms,num_of_writes,num_of_bytes_written,io_stall_write_ms)
    SELECT database_id,file_id,num_of_reads,num_of_bytes_read
        ,io_stall_read_ms,num_of_writes,num_of_bytes_written,io_stall_write_ms
    FROM sys.dm_io_virtual_file_stats(NULL,NULL)
OPTION (RECOMPILE);

-- Set test interval (1 minute).
WAITFOR DELAY '00:01:00.000';

;WITH Stats(db_id, file_id, Reads, ReadBytes, Writes
    ,WrittenBytes, ReadStall, WriteStall)
as
(
    SELECT
        s.database_id, s.file_id
        ,fs.num_of_reads – s.num_of_reads
        ,fs.num_of_bytes_read – s.num_of_bytes_read
        ,fs.num_of_writes – s.num_of_writes
        ,fs.num_of_bytes_written – s.num_of_bytes_written
        ,fs.io_stall_read_ms – s.io_stall_read_ms
        ,fs.io_stall_write_ms – s.io_stall_write_ms
    FROM
        #Snapshot s JOIN sys.dm_io_virtual_file_stats(NULL, NULL) fs ON
            s.database_id = fs.database_id and s.file_id = fs.file_id
)
SELECT
    s.db_id AS [DB ID], d.name AS [Database]
    ,mf.name AS [File Name], mf.physical_name AS [File Path]
    ,mf.type_desc AS [Type], s.Reads
    ,CONVERT(DECIMAL(12,3), s.ReadBytes / 1048576.) AS [Read MB]
    ,CONVERT(DECIMAL(12,3), s.WrittenBytes / 1048576.) AS [Written MB]
    ,s.Writes, s.Reads + s.Writes AS [IO Count]
    ,CONVERT(DECIMAL(5,2),100.0 * s.ReadBytes /
            (s.ReadBytes + s.WrittenBytes)) AS [Read %]
    ,CONVERT(DECIMAL(5,2),100.0 * s.WrittenBytes /
            (s.ReadBytes + s.WrittenBytes)) AS [Write %]
    ,s.ReadStall AS [Read Stall]
    ,s.WriteStall AS [Write Stall]
    ,CASE WHEN s.Reads = 0
```

```
        THEN 0.000
        ELSE CONVERT(DECIMAL(12,3),1.0 * s.ReadStall / s.Reads)
    END AS [Avg Read Stall]
    ,CASE WHEN s.Writes = 0
        THEN 0.000
        ELSE CONVERT(DECIMAL(12,3),1.0 * s.WriteStall / s.Writes)
    END AS [Avg Write Stall]
FROM
    Stats s JOIN sys.master_files mf WITH (NOLOCK) ON
        s.db_id = mf.database_id and
        s.file_id = mf.file_id
    JOIN sys.databases d WITH (NOLOCK) ON
        s.db_id = d.database_id
WHERE -- Only display files with more than 20MB throughput
    (s.ReadBytes + s.WrittenBytes) > 20 * 1048576
ORDER BY
    s.db_id, s.file_id
OPTION (RECOMPILE);
```

На рис. 3.4 показан вывод этого представления.

Нужно, чтобы показатели задержек (stalls) оставались как можно ниже. Невозможно определить пороговые значения, которые подходили бы ко всем системам, но я обычно стараюсь, чтобы задержки записи журналов транзакций не превышали 1–2 мс, а задержки чтения и записи файлов данных — 3–5 мс, если используется сетевое хранилище. Если применять современные накопители с прямым подключением, то задержки могут стать даже меньше миллисекунды.

Затем проанализируйте пропускную способность системы. Высокие задержки в сочетании с низкой пропускной способностью обычно указывают на проблемы, не связанные непосредственно с SQL Server. Не забудьте изучить пропускную способность для всех файлов, которые используют один и тот же диск или контроллер. Высокая пропускная способность для одних файлов может повлиять на показатели других файлов, использующих тот же ресурс.

Обычно пропускная способность коррелирует с задержками: чем больше данных читается или записывается, тем выше задержка. Эта корреляция обычно линейна, пока не будет достигнут переломный момент, после которого задержка быстро увеличивается.

Большое количество операций чтения и задержек чтения в файлах данных часто сопровождается значительным процентом ожиданий `PAGEIOLATCH` и низкой ожидаемой продолжительностью существования страниц (`Page Life Expectancy`). Это признак того, что с диска постоянно считываются большие объемы данных. Нужно разобраться, почему это происходит. В большинстве случаев это связано с неоптимизированными запросами, которые злоупотребляют слишком объемными операциями просмотра при чтении данных с диска. Как обнаружить эти запросы, мы поговорим в главе 4.

	DB ID	Database	File Name	File Path	Type	Reads	Read MB	Written MB	Writes
1	2	tempdb	tempdev	T:\SQLDa…	ROWS	279	17.375	17.563	281
2	2	tempdb	templog	T:\SQLDa…	LOG	0	0.000	76.727	1311
3	2	tempdb	temp2	T:\SQLDa…	ROWS	278	17.086	17.211	276
4	2	tempdb	temp3	T:\SQLDa…	ROWS	292	17.836	18.000	290
5	2	tempdb	temp4	T:\SQLDa…	ROWS	295	18.031	18.273	294
6	2	tempdb	temp5	T:\SQLDa…	ROWS	307	18.617	18.742	301
7	2	tempdb	temp6	T:\SQLDa…	ROWS	287	17.664	17.914	287
8	2	tempdb	temp7	T:\SQLDa…	ROWS	284	17.625	17.750	284
9	11				ROWS	76	1.578	88.719	7551
10	11				LOG	0	0.000	142.578	31166
11	11				ROWS	1111	9.453	38.422	4420
12	11				ROWS	2505	44.227	345.977	37251

	IO Count	Read %	Write %	Read Stall	Write Stall	Avg Read Stall	Avg Write Stall
1	560	49.73	50.27	205	322	0.735	1.146
2	1311	0.00	100.00	0	1191	0.000	0.908
3	554	49.82	50.18	195	308	0.701	1.116
4	582	49.77	50.23	206	322	0.705	1.110
5	589	49.67	50.33	206	323	0.698	1.099
6	608	49.83	50.17	225	343	0.733	1.140
7	574	49.65	50.35	214	308	0.746	1.073
8	568	49.82	50.18	208	299	0.732	1.053
9	7627	1.75	98.25	60	6847	0.789	0.907
10	31166	0.00	100.00	0	16944	0.000	0.544
11	5531	19.75	80.25	1994	3806	1.795	0.861
12	39756	11.33	88.67	2252	29908	0.899	0.803

Рис. 3.4. Пример вывода представления `sys.dm_io_virtual_file_stats`

Не отвергайте и вероятность того, что серверу не хватает ресурсов и на нем недостаточно памяти, чтобы разместить активный набор данных. В любом случае добавить дополнительную память — вполне приемлемый способ снизить нагрузку на ввод/вывод и повысить производительность системы. Очевидно, что этот способ — не наилучший, но часто бывает проще и дешевле решить проблему с помощью аппаратных средств.

В базах данных пользователей большое количество операций записи и задержек записи в файлы данных часто свидетельствует о том, что контрольные точки настроены неэффективно. Можно добиться улучшений, если отрегулировать конфигурацию контрольных точек, как я покажу позже в этой главе. В долгосрочной перспективе стоит проанализировать, получится ли уменьшить ко-

личество страниц данных, которые SQL Server записывает на диск. Для этого можно удалить ненужные индексы, сократить количество разбиений страниц (используйте для этого параметр FILLFACTOR, а также настройте стратегию обслуживания индекса), уменьшить количество страниц данных за счет сжатия данных и, возможно, провести рефакторинг схемы базы данных и приложений.

Если высокая пропускная способность сопровождается задержками в базе данных tempdb — определите, что их вызывает. Три наиболее распространенные причины — это активность хранилища версий, массовые переносы в базу данных tempdb и злоупотребление временными объектами. Я расскажу об этом в главе 9.

Наконец, можно получить представление о задержке ввода/вывода, изучив время ожидания ресурса в PAGEIOLATCH и другие ожидания, связанные с вводом/выводом. При этом вы не получите подробной информации по каждому файлу, но это полезно, чтобы оценить производительность ввода/вывода в масштабе всей системы.

Счетчики производительности и метрики ОС

Представление sys.dm_io_virtual_file_stats содержит полезную и подробную информацию и подсказывает направление для дальнейшего устранения неполадок ввода/вывода, но у него есть ограничение: оно усредняет данные за некоторый интервал времени.

Это приемлемо, когда задержка ввода/вывода невелика. Однако если значения задержки высоки, то нужно определить, тормозит система в принципе или данные искажены из-за отдельных всплесков активности. Для этого можно просмотреть метрики, показывающие связь между производительностью SQL Server и диска.

Процесс устранения неполадок в Windows и Linux немного различается. В Windows самое простое средство анализа метрик — известная утилита PerfMon (Монитор ресурсов). Можно параллельно просматривать счетчики производительности SQL Server и ввода/вывода и сопоставлять их показатели.

В табл. 3.1 приведены счетчики производительности, которые нужно проанализировать.

Первым делом я обычно смотрю на задержки Avg Disk sec/Read и Avg Disk sec/Write, а также Avg Disk Queue Length. Если на графиках этих показателей наблюдаются пики, то я добавляю счетчики, специфичные для SQL Server, чтобы понять, какие процессы приводят ко всплескам активности.

На рис. 3.5 приведен пример, где можно увидеть корреляцию между Checkpoint pages/sec, Avg Disk sec/Write и Avg Disk Queue Length. Отсюда легко заключить, что подсистема ввода/вывода не справляется со всплесками операций записи, вызванных процессом контрольной точки.

Таблица 3.1. Счетчики производительности, связанные с вводом/выводом

Объект	Счетчики производительности	Описание
Физический диск	Avg Disk Queue Length Avg Disk Read Queue Length Avg Disk Write Queue Length	Среднее количество запросов ввода/вывода (всего, чтения и записи соответственно), поставленных в очередь в течение выборочного интервала. Эти показатели должны быть как можно ниже. Пики соответствуют запросам ввода/вывода, которые ставятся в очередь на уровне ОС
	Current Disk Queue Length	Длина очереди ввода/вывода в момент измерения
	Avg Disk sec/Transfer Avg Disk sec/Read Avg Disk sec/Write	Средняя задержка дисковых операций за выборочный интервал времени. Эти цифры, как правило, аналогичны показателям задержки из представления `sys.dm_io_virtual_file_stats` за тот же период. Однако sys.dm_io_virtual_file_stats обычно измеряется за большие интервалы времени, а перечисленные здесь показатели дают понять, типичны ли высокие задержки в целом или они проявляются отдельными пиками
	Disk Transfers/sec Disk Reads/sec Disk Writes/sec Disk Bytes/sec Disk Read Bytes/sec Disk Write Bytes/sec	Количество операций ввода/вывода и пропускная способность на момент считывания. Как и показатели задержки, эти метрики показывают, равномерно ли по времени распределена нагрузка на диск
	Avg Disk Bytes/Transfer Avg Disk Bytes/Read Avg Disk Bytes/Write	Средний размер запросов ввода/вывода; позволяет анализировать характерные схемы ввода/вывода в системе
SQL Server: Диспетчер буфера	Checkpoint pages/sec Background writer pages/sec	Число «грязных» страниц, записанных процессом контрольных точек
	Lazy writer/sec	Число страниц, записанных процессом отложенной записи
	Page reads/sec Page writes/sec	Количество физических операций чтения и записи
	Readahead pages/sec	Число страниц, прочитанных процессом упреждающего чтения

Объект	Счетчики производительности	Описание

SQL Server: Базы данных	`Log Bytes Flushed/sec`
	`Log Flush Write Time (ms)`
	`Log Flushes/sec`
		Пропускная способность, задержки и число запросов на запись журналов транзакций. В случае задержек протоколирования эти счетчики помогают понять, равномерно ли генерируются журналы

SQL Server: Статистика SQL	`Batch Requests/sec`	Эти два показателя не связаны с вводом/выводом напрямую, но их можно использовать, чтобы анализировать пики рабочей нагрузки, которые, в свою очередь, могут привести к всплескам активности ввода/вывода
SQL Server: Базы данных	`Transactions/sec`	

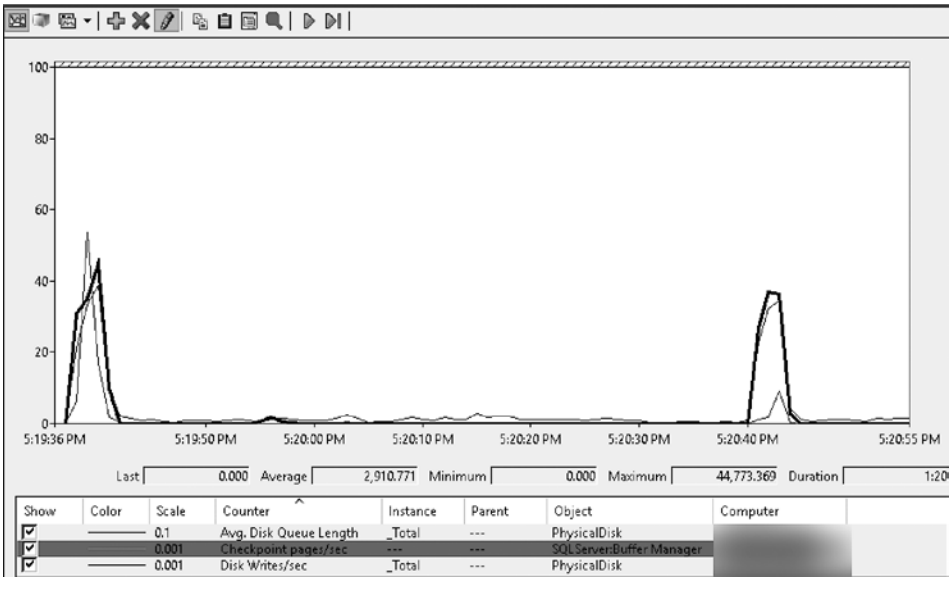


Рис. 3.5. Контрольные точки и дисковая очередь

Обратите внимание, какие еще приложения установлены на сервере. Возможно, именно они виноваты во всплесках активности ввода/вывода или других проблемах.

В Linux нет утилиты `PerfMon`, зато доступно множество других бесплатных и коммерческих средств мониторинга. Еще можно использовать такие инструменты, как `iostat`, `dstat` и `iotop`, которые включены в основные дистрибутивы Linux.

Они позволяют оценить показатели производительности диска для каждого процесса или всей системы.

На стороне SQL Server счетчики производительности можно посмотреть в представлении `sys.dm_os_performance_counters`[1]. В листинге 3.2 показано, как это сделать. Измените префикс `SQLserver` в столбце `object_name` на `MSSQL$<имя_вашего_экземпляра>`, если вы запускаете код на именованном экземпляре SQL Server.

Листинг 3.2. Использование представления `sys.dm_os_performance_counters`

```
DROP TABLE IF EXISTS #PerfCntrs;
GO

CREATE TABLE #PerfCntrs
(
    collected_time DATETIME2(7) NOT NULL DEFAULT SYSDATETIME(),
    object_name SYSNAME NOT NULL,
    counter_name SYSNAME NOT NULL,
    instance_name SYSNAME NOT NULL,
    cntr_value BIGINT NOT NULL,
    PRIMARY KEY (object_name, counter_name, instance_name)
);

;WITH Counters(obj_name, ctr_name)
AS
(
    SELECT C.obj_name, C.ctr_name
    FROM
    (
        VALUES
            ('SQLServer:Buffer Manager','Checkpoint pages/sec')
            ,('SQLServer:Buffer Manager','Background writer pages/sec')
            ,('SQLServer:Buffer Manager','Lazy writes/sec')
            ,('SQLServer:Buffer Manager','Page reads/sec')
            ,('SQLServer:Buffer Manager','Page writes/sec')
            ,('SQLServer:Buffer Manager','Readahead pages/sec')
            ,('SQLServer:Databases','Log Flushes/sec') -- For all DBs
            ,('SQLServer:Databases','Log Bytes Flushed/sec') -- For all DBs
            ,('SQLServer:Databases','Log Flush Write Time (ms)') -- For all DBs
            ,('SQLServer:Databases','Transactions/sec') -- For all DBs
            ,('SQLServer:SQL Statistics','Batch Requests/sec')
    ) C(obj_name, ctr_name)
)
INSERT INTO #PerfCntrs(object_name,counter_name,instance_name,cntr_value)
    SELECT
        pc.object_name, pc.counter_name, pc.instance_name, pc.cntr_value
    FROM
        sys.dm_os_performance_counters pc WITH (NOLOCK) JOIN Counters c ON
```

[1] https://oreil.ly/DJ4f5

```
            pc.counter_name = c.ctr_name AND pc.object_name = c.obj_name;
WAITFOR DELAY '00:00:01.000';

;WITH Counters(obj_name, ctr_name)
AS
(
    SELECT C.obj_name, C.ctr_name
    FROM
    (
        VALUES
            ('SQLServer:Buffer Manager','Checkpoint pages/sec')
            ,('SQLServer:Buffer Manager','Background writer pages/sec')
            ,('SQLServer:Buffer Manager','Lazy writes/sec')
            ,('SQLServer:Buffer Manager','Page reads/sec')
            ,('SQLServer:Buffer Manager','Page writes/sec')
            ,('SQLServer:Buffer Manager','Readahead pages/sec')
            ,('SQLServer:Databases','Log Flushes/sec') -- For all DBs
            ,('SQLServer:Databases','Log Bytes Flushed/sec') -- For all DBs
            ,('SQLServer:Databases','Log Flush Write Time (ms)') -- For all DBs
            ,('SQLServer:Databases','Transactions/sec') -- For all DBs
            ,('SQLServer:SQL Statistics','Batch Requests/sec')
    ) C(obj_name, ctr_name)
)
SELECT
    pc.object_name, pc.counter_name, pc.instance_name
    ,CASE pc.cntr_type
        WHEN 272696576 THEN
            (pc.cntr_value – h.cntr_value) * 1000 /
                DATEDIFF(MILLISECOND,h.collected_time,SYSDATETIME())
        WHEN 65792 THEN
            pc.cntr_value
        ELSE NULL
    END as cntr_value
FROM
    sys.dm_os_performance_counters pc WITH (NOLOCK) JOIN Counters c ON
        pc.counter_name = c.ctr_name AND pc.object_name = c.obj_name
    JOIN #PerfCntrs h ON
        pc.object_name = h.object_name AND
        pc.counter_name = h.counter_name AND
        pc.instance_name = h.instance_name
ORDER BY
    pc.object_name, pc.counter_name, pc.instance_name
OPTION (RECOMPILE);
```

Можно также добавить к анализу представление `sys.dm_io_virtual_file_stats`, собирая из него данные и счетчики производительности ежесекундно. Подход будет аналогичен тому, который мы только что обсуждали: изучить корреляцию между задержкой диска и активностью операций и оценить общую производительность подсистемы ввода/вывода, определяя критические точки в нагрузке.

Виртуализация, хост-адаптер шины и уровни хранения

В стеке хранилища имеет смысл проанализировать еще несколько уровней, помимо ОС. Это уровень виртуализации, конфигурация контроллера HBA/SCSI и само физическое устройство хранения.

SQL Server, как правило, работает в общих средах. Он делит устройства хранения и сетевую инфраструктуру с другими клиентами, а при виртуализации работает на одном физическом хосте с другими виртуальными машинами. Как я уже говорил ранее в этой книге, при использовании виртуализации следует убедиться, что хост не перегружен, иначе вы рискуете столкнуться с самыми разными проблемами производительности.

За исключением самых простых конфигураций SQL Server, использующих локальное хранилище, запросы ввода/вывода сериализуются и отправляются по сети. Здесь бывают две типичные проблемы.

Первая — *недостаточная емкость очереди* где-то на пути ввода/вывода. К сожалению, вместимости очереди по умолчанию может не хватить для очень ресурсоемкой рабочей нагрузки ввода/вывода. Вам нужно будет проверить и, возможно, увеличить этот параметр в настройках хранилища данных, контроллера vSCSI и хост-адаптера шины (HBA). Типичный признак недостаточной емкости очереди — низкая задержка в хранилище в сочетании с гораздо более высокой задержкой в виртуальной машине и/или ОС при наличии дисковой очереди.

Вторая проблема — *«шумные соседи»*. Когда на одном хосте работает несколько виртуальных машин с интенсивным вводом/выводом, они могут влиять друг на друга. Аналогично, если несколько серверов с высокой пропускной способностью используют общую сеть и общее хранилище, эти ресурсы могут оказаться перегружены. К сожалению, устранение неполадок, связанных с «шумными соседями», — это всегда сложная задача, в рамках которой приходится анализировать многие компоненты инфраструктуры.

У каждого хранилища есть ограничение на количество ожидающих запросов, которые оно может обработать. Если увеличить емкость очереди на высоконагруженном сервере, от этого может вырасти число ожидающих запросов в хранилище. Таким образом вы просто перенесете узкое место с одного уровня на другой, особенно если хранилище обслуживает запросы от большого числа высоконагруженных систем.

Хост виртуализации и хранилище предоставляют данные о своей пропускной способности, числе операций ввода/вывода в секунду и задержках. На уровнях виртуализации эти метрики могут различаться в зависимости от используемой технологии. Например, в Hyper-V можно использовать обычные показатели

производительности диска на хосте. В VMware можно получить данные с помощью утилиты ESXTOP, о которой я расскажу в главе 15. В любом случае подход к устранению неполадок очень похож на то, что мы уже обсуждали: нужно изучить доступные метрики, сопоставить их показания и найти узкие места на пути ввода/вывода.

Наконец, проверьте конфигурацию хранилища. Производители устройств хранения обычно публикуют рекомендации о том, как настраивать SQL Server для работы с их продукцией: эти рекомендации послужат хорошей отправной точкой. Однако обратите внимание на то, чтобы размер единицы распределения дискового пространства согласовывался с шириной полосы чередования RAID и смещением раздела.

Например, смещение раздела 1024 Мбайт, блок диска 4 Кбайт, единица распределения 64 Кбайт и полосы RAID по 128 Кбайт идеально согласуются друг с другом, и при этом каждый запрос ввода/вывода обслуживается одним диском. Однако полосы RAID по 96 Кбайт будут распределять 64-килобайтовые единицы по двум дискам, что приведет к дополнительным запросам ввода/вывода и может серьезно ухудшить производительность.

Снова подчеркнем, что *всегда* полезно работать совместно со специалистами по инфраструктуре и устройствам хранения. Будучи экспертами в своей предметной области, они могут помочь вам найти основную причину проблемы быстрее, чем если вы работаете в одиночку.

Наконец, лучший способ добиться предсказуемой производительности в критически важных системах — это выделенная среда. Запускайте SQL Server на выделенном оборудовании с напрямую подключенными устройствами хранения, чтобы максимизировать производительность.

Настройка контрольных точек

Как мы уже знаем, в SQL Server используется протоколирование с опережающей записью. Транзакции считаются принятыми только после того, как в журналах транзакций появятся соответствующие записи. SQL Server не обязательно одновременно с этим сохраняет «грязные» страницы данных на диск, потому что при необходимости он может повторно применить изменения, воспроизведя записи журнала.

Процесс контрольных точек сохраняет страницы данных в файлы данных. Основная задача контрольной точки — сократить время восстановления в случае сбоя SQL или аварийного переключения: чем меньше изменений приходится воспроизводить, тем быстрее выполнится восстановление. Максимальное *желаемое* время восстановления контролируется на уровне либо сервера, либо базы данных. По умолчанию и то и другое равно 60 секундам.

Не рассматривайте целевое время восстановления как жестко заданное значение. Часто база данных будет восстанавливаться гораздо быстрее. И наоборот, всплески активности и длительные транзакции могут продлить время восстановления.

Контрольные точки бывают четырех различных типов.

Внутренние контрольные точки

Возникают во время некоторых операций SQL Server, таких как запуск резервного копирования базы данных или создание моментального снимка базы данных.

Ручные контрольные точки

Возникают, когда пользователи создают их с помощью команды `CHECKPOINT`.

Автоматические контрольные точки

Исторически в SQL Server использовались автоматические контрольные точки, а интервал восстановления контролировался сервером. Процесс контрольной точки активируется один или несколько раз в течение каждого интервала восстановления и сбрасывает «грязные» страницы данных на диск. К сожалению, такой подход может привести к всплескам записи данных, что чревато проблемами в высоконагруженных системах.

Косвенные контрольные точки

Этот метод, доступный в SQL Server 2012 и более поздних версиях, пытается сбалансировать нагрузку ввода/вывода, создавая контрольные точки гораздо чаще, а иногда и вовсе непрерывно. Это помогает смягчить всплески записи данных и уравновесить нагрузку ввода/вывода. Используйте именно этот тип вместо автоматических контрольных точек всегда, когда возможно.

Косвенная контрольная точка управляется каждой базой данных отдельно и по умолчанию применяется в базах, созданных в SQL Server 2016 и более поздних версиях. Но SQL Server *не* активирует косвенную контрольную точку автоматически при обновлении экземпляра SQL Server и в версиях SQL Server 2012 и 2014. Ее можно включить вручную, настроив целевое время восстановления на уровне базы данных с помощью команды `ALTER DATABASE SET TARGET_RECOVERY_TIME`.

Вот пример из одной системы, с которой я работал. Выборка данных из представления `sys.dm_io_virtual_file_stats` за 1 минуту показывала очень большую задержку записи для файлов данных. Однако выборки меньшей продолжительности (от 1 до 3 секунд) вообще редко проявляли какую-либо активность.

На рис. 3.6 показаны соответствующие данные: вверху — выборка продолжительностью в 1 минуту, внизу — в 1 секунду.

	File Path	Reads	Read MB	Written MB	Writes	IO Count	Read Stall
1	File1.NDF	3098	91.867	1404.727	138564	141622	11708
2	File2.NDF	47398	2104.140	9648.797	1113960	1161098	466594

Write Stall	Avg Read Stall	Avg Write Stall
8287158	3.779	59.807
74303099	9.773	66.750

	File Path	Reads	Read MB	Written MB	Writes	IO Count	Read Stall
1	File1.NDF	14	0.641	0.000	0	14	14
2	File2.NDF	552	27.399	0.000	0	552	471

Write Stall	Avg Read Stall	Avg Write Stall
1.000	0.000	0.000
0.000	0.853	0.000

Рис. 3.6. Выборочные данные представления `sys.dm_io_virtual_file_stats` с автоматической контрольной точкой

Такое поведение навело меня на мысль о том, что проблема связана с контрольной точкой. Гипотеза подтвердилась, когда я взглянул на счетчики производительности `Checkpoint page/sec`, `Disk Writes/sec` и `Avg Disk Queue Length`. На рис. 3.5, приведенном ранее в этой главе, показан график из `PerfMon`, на котором хорошо виден всплеск записи на диск от процесса контрольной точки.

Хотя в этом примере использовался SQL Server 2016, в нем применялась автоматическая контрольная точка, потому что все базы данных были обновлены с более ранней версии SQL Server. Когда я включил косвенную контрольную точку, схема ввода/вывода сразу же стала гораздо более сбалансированной.

Новые метрики производительности показаны на рис. 3.7. Обратите внимание, что в случае с косвенной контрольной точкой нужно смотреть на счетчик `Background writer pages/sec`, а не `Checkpoint pages/sec`.

На рис. 3.8 показаны метрики выборки продолжительностью 1 минуту в представлении `sys.dm_io_virtual_file_stats`. Как видите, задержка пришла в норму.

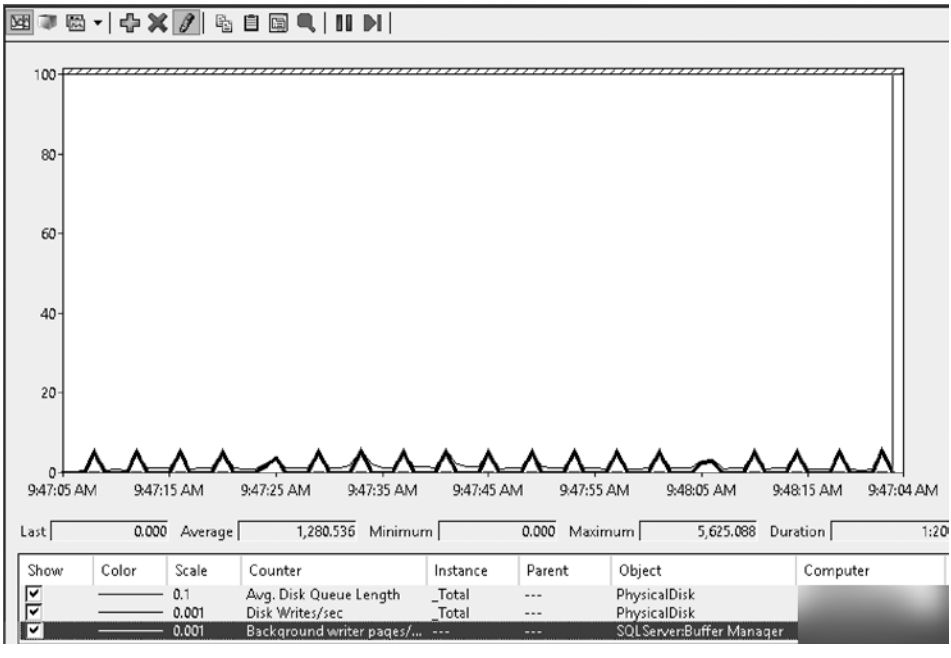


Рис. 3.7. Метрики производительности косвенных контрольных точек

	File Path	Reads	Read MB	Written MB	Writes	IO Count
1	File1.NDF	2987	89.121	1367.542	135039	138026
2	File2.NDF	48212	2140.134	10944.122	1182783	1230995

Read Stall	Write Stall	Avg Read Stall	Avg Write Stall
3065	264675	1.026	1.960
66195	2280405	1.373	1.928

Рис. 3.8. Пример представления `sys.dm_io_virtual_file_stats` с косвенной контрольной точкой

Косвенные контрольные точки не избавляют от всех всплесков ввода/вывода. Всплески все равно могут происходить, особенно если в системе наблюдаются пики модификации данных. Однако все равно всплески встречаются намного реже, чем в случае автоматических контрольных точек.

Целевое время восстановления нужно настроить так, чтобы максимально сбалансировать нагрузку ввода/вывода. В предыдущем случае я добился наилучших результатов, задав время 90 секунд. Разумеется, при более высоких значениях может увеличиться время восстановления.

Ожидания ввода/вывода

В SQL Server с операциями ввода/вывода связано несколько типов ожидания. Иногда они проявляются все сразу, если дисковая подсистема работает недостаточно быстро. Рассмотрим пять самых распространенных из них: `ASYNC_IO_COMPLETION`, `IO_COMPLETION`, `WRITELOG`, `WRITE_COMPLETION` и `PAGEIOLATCH`.

Ожидание ASYNC_IO_COMPLETION

Этот тип ожидания возникает, когда SQL Server ожидает завершения асинхронных операций ввода/вывода (чтения или записи) для страниц не из буферного пула. Например:

Обычные контрольные точки

Внутренние контрольные точки, возникающие, когда запускается резервное копирование базы данных или команда `DBCC CHECKDB`

Чтение страниц глобальной карты распределения (GAM) из файлов данных

Чтение страниц данных из базы данных во время ее резервного копирования (к сожалению, эта операция искажает среднее время ожидания и затрудняет анализ).

Когда я вижу много ожиданий `ASYNC_IO_COMPLETION` и `PAGEIOLATCH`, то выполняю общий поиск неполадок ввода/вывода. Если ожиданий `PAGEIOLATCH` нет, то я смотрю, как часто возникает `ASYNC_IO_COMPLETION`. Это ожидание можно игнорировать, если его процент не очень значителен, а задержка диска мала.

Ожидание IO_COMPLETION

Ожидание `IO_COMPLETION` возникает во время синхронного чтения и записи в файлы данных и во время некоторых операций чтения из журнала транзакций. Вот несколько примеров:

- Чтение страниц карты распределения из базы данных.
- Чтение журнала транзакций во время восстановления базы данных.
- Запись данных в `tempdb` во время переносов сортировки.

Если наблюдается значительная доля этого типа ожидания, выполните общую диагностику производительности диска. Обратите особое внимание на задержку и пропускную способность `tempdb`. По моему опыту, плохая производительность `tempdb` служит наиболее распространенной причиной такого ожидания. Подробнее об устранении неполадок `tempdb` я расскажу в главе 9.

Ожидание WRITELOG

Как можно догадаться по названию, это ожидание происходит, когда SQL Server вносит записи в журнал транзакций. Оно появляется в любой системе, и это нормально, но слишком большой его процент может указывать на узкое место в журнале транзакций.

Посмотрите на среднее время ожидания и задержку записи в журнал транзакций в представлении `sys.dm_io_virtual_file_stats`. Высокие значения могут влиять на пропускную способность системы.

Чтобы сократить ожидания этого типа, можно оптимизировать пропускную способность дисковой подсистемы, а также сделать еще несколько вещей, которые мы обсудим в главе 11.

Ожидание WRITE_COMPLETION

Это ожидание происходит во время синхронных операций записи в файлы базы данных и журналов. По моему опыту, чаще всего оно появляется при снимках базы данных.

Чтобы поддерживать моментальные снимки базы, SQL Server сохраняет версии страниц данных, которые существовали на момент снимка. В контрольной точке SQL Server считывает старые копии страниц данных из файлов данных и сохраняет их в виде снимка перед тем, как записывать измененные страницы на диск. Это может значительно увеличить количество операций ввода/вывода в системе.

Если появляется такое ожидание, проверьте наличие моментальных снимков базы данных. Учтите, что некоторые внутренние процессы, такие как `DBCC CHECKDB`, тоже создают снимки.

Если в системе есть моментальные снимки БД и они действительно нужны, то следует подумать, как улучшить производительность диска. В прочих случаях снимки можно удалить, если из-за них тормозят устройства хранения.

Ожидания PAGEIOLATCH

Как вы уже знаете, ожидания `PAGEIOLATCH` происходят, когда SQL Server считывает страницы данных с диска. Эти ожидания очень распространены и встречаются в каждой системе. Формально существуют шесть ожиданий такого типа, но реально заметны только три из них:

`PAGEIOLATCH_EX`

Происходит, когда рабочий процесс хочет обновить страницу данных и ожидает, пока она будет прочитана с диска в буферный пул.

PAGEIOLATCH_SH

Происходит, когда рабочий процесс хочет прочитать страницу данных и ожидает, пока она будет прочитана с диска в буферный пул.

PAGEIOLATCH_UP

Происходит, когда рабочий процесс хочет обновить системную страницу (например, карту распределения) и ожидает, пока она будет прочитана с диска в буферный пул.

Чрезмерное количество ожиданий PAGEIOLATCH говорит о том, что SQL Server постоянно читает данные с диска. Обычно это происходит в двух случаях.

Первый случай — это недостаточно мощный SQL Server, где активные данные не помещаются в памяти.

Второй и более распространенный случай — неоптимизированные запросы, которые просматривают ненужные данные, очищая содержимое буферного пула.

Чтобы перепроверить данные, можно взглянуть на счетчик производительности Page Life Expectancy, который показывает, как долго страницы данных хранятся в буферном пуле. В качестве ориентира можно взять значение 300 с на каждые 4 Гбайт памяти буферного пула: например, 7500 с на сервере с буферным пулом 100 Гбайт.

Значение Page Life Expectancy можно узнать с помощью утилиты PerfMon или представления sys.dm_os_performance_counters, как показано в листинге 3.3. Это же представление возвращает показатели для отдельных узлов NUMA.

Листинг 3.3. Получение значения Page Life Expectancy

```
SELECT object_name, counter_name, instance_name, cntr_value as [PLE(sec)]
FROM sys.dm_os_performance_counters WITH (NOLOCK)
WHERE counter_name = 'Page life expectancy'
OPTION (RECOMPILE);
```

Большой процент ожиданий PAGEIOLATCH всегда нужно устранять. С одной стороны, ожидания этого типа не обязательно создают проблемы для клиентов, особенно с дисковыми массивами на основе флеш-памяти с низкой задержкой. Но с другой стороны, если данные разрастутся так, что перестанут умещаться на диск, это быстро скажется на всей системе.

Чтобы уменьшить влияние ожиданий PAGEIOLATCH, можно обновить дисковую подсистему или добавить на сервер больше памяти. Однако лучше всего обнаружить и оптимизировать неэффективные запросы, чтобы уменьшить объем данных, считываемых с диска. В следующей главе вы узнаете, как выявлять эти запросы.

Резюме

В SQL Server используются совместное планирование и в большинстве случаев — асинхронный ввод/вывод для чтения и записи данных. По умолчанию каждый планировщик обрабатывает ввод/вывод и имеет собственную очередь ввода/вывода.

В представлении `sys.dm_io_virtual_file_stats` можно найти показатели пропускной способности ввода/вывода и задержки для каждого файла базы данных. В правильно настроенной системе с сетевым хранилищем задержка записи журнала транзакций не должна превышать 1–2 мс, а задержка чтения и записи файлов данных — 3–5 мс. При прямом подключении хранилища (DAS) эти задержки должны быть еще ниже.

Устраняя неполадки с производительностью ввода/вывода, изучите весь стек ввода/вывода. Проблема может оказаться где угодно: в ОС, виртуализации, сетевом пути или на уровне устройств хранения.

Зачастую высокие задержки возникают из-за всплесков активности ввода/вывода. Проанализируйте и настройте процесс контрольных точек. Это одна из самых распространенных ситуаций в высоконагруженных системах. Во многих случаях снижение активности диска поможет уменьшить его задержку и производительность системы. Оптимизация запросов — один из лучших способов добиться этого. В следующей главе мы рассмотрим, как выявлять неоптимизированные запросы.

Чек-лист устранения неполадок

- ☐ Проанализировать задержки дисковой подсистемы в представлении `sys.dm_io_virtual_file_stats`.
- ☐ Проверить, не вызвана ли высокая задержка всплесками активности ввода/вывода, изучив счетчики производительности SQL Server и ОС.
- ☐ Проверить показатели ввода/вывода на уровне виртуальной машины и хранилища, обращая внимание на «шумных соседей».
- ☐ Проверить настройки емкости дисковой очереди в стеке ввода/вывода.
- ☐ Устранить неполадки с производительностью контрольных точек SQL Server и настроить косвенные контрольные точки.
- ☐ Устранить проблемы с производительностью журнала, если наблюдаются значительные ожидания `WRITELOG` (см. главу 11).
- ☐ Устранить проблемы с производительностью базы данных `tempdb`, если наблюдается много ожиданий `IO_COMPLETION`, а база `tempdb` используется чересчур активно и вызывает задержки (см. главу 9).
- ☐ Выявить и оптимизировать неэффективные запросы, если наблюдаются значительные ожидания `PAGEIOLATCH`.

ГЛАВА 4

Неэффективные запросы

Неэффективные запросы существуют в любой системе. Их воздействие на производительность проявляется по-разному: в первую очередь это увеличение нагрузки ввода/вывода, использования ЦП и блокировок. Такие запросы важно обнаружить и оптимизировать.

В этой главе обсуждается, как неэффективные запросы влияют на систему, а также даны рекомендации, как их обнаружить. В первую очередь рассмотрим, как для этого использовать статистику выполнения на основе кэша планов. Затем мы изучим расширенные события (Extended Events), трассировку SQL (SQL Traces) и хранилище запросов (Query Store), а закончим главу замечаниями о сторонних инструментах мониторинга. В следующих главах мы разберем стратегии оптимизации неэффективных запросов.

Чем плохи неэффективные запросы

За время своей карьеры в области баз данных я еще не видел системы, где оптимизация запросов не пошла бы на пользу. Я уверен, что такие совершенные примеры существуют, но меня не зовут анализировать системы, где все в порядке. В любом случае, идеальных систем очень мало, а во всех остальных есть что улучшать и оптимизировать.

Не каждую компанию вообще заботит оптимизация запросов. Это трудоемкий и утомительный процесс. Чтобы быстрее разработать и вывести продукт на рынок, во многих случаях проще добавить аппаратных мощностей, чем тратить долгие часы, копаясь в запросах.

Но с определенного момента такой подход приводит к проблемам с масштабируемостью. Плохо оптимизированные запросы влияют на систему со многих сторон, но самый очевидный пример — это производительность диска. Если подсистема ввода/вывода не справляется с нагрузкой объемных операций просмотра, то ухудшается производительность всей системы.

До некоторой степени эту проблему можно замаскировать, добавив на сервер больше памяти. Это увеличит размер буферного пула и позволит SQL Server кэшировать больше данных, сокращая физический ввод/вывод. Но по мере того, как объем данных в системе растет, такой подход может стать непрактичным или даже невозможным, особенно в отличных от Enterprise выпусках SQL Server, где максимальный размер буферного пула ограничен.

Еще один эффект заключается в том, что неоптимизированные запросы сильно нагружают ЦП на серверах. Чем больше данных обрабатывается, тем больше ресурсов ЦП потребляется. На одну операцию логического чтения страницы данных или ее просмотра в памяти у сервера может уходить всего несколько микросекунд, но суммарные затраты времени на все операции быстро растут, когда количество операций увеличивается.

Опять же, можно замаскировать проблему, добавив на сервер больше процессоров. (Хотя при этом придется платить за дополнительные лицензии, причем в версиях, отличных от Enterprise, максимальное количество ЦП ограничено.) Более того, дополнительные ЦП не всегда помогают, потому что неоптимизированные запросы все равно будут вызывать блокировку. Существуют способы уменьшить блокировку без тонкой настройки запросов, но такие меры могут повлиять на работу системы в целом и ее производительность.

Суть такова: когда вы устраняете неполадки, всегда анализируйте систему на предмет неоптимизированных запросов. После этого оцените, насколько сильно такие запросы влияют на систему.

Хотя оптимизация запросов всегда идет на пользу, это сложное занятие и усилия не обязательно окупаются. Но в большинстве случаев имеет смысл привести в порядок хотя бы некоторые запросы.

К примеру, я перерабатываю запросы, когда вижу большую загруженность диска, блокировки или высокую нагрузку ЦП. Но если данные кэшируются в буферном пуле, а нагрузка ЦП приемлема, то я предпочитаю сперва сосредоточиться на других очагах проблем. Тем не менее я не теряю бдительности и учитываю, что может произойти по мере роста объема данных. Не исключено, что в один прекрасный момент активные данные перестанут помещаться в буферный пул, что приведет к внезапному и серьезному падению производительности.

К счастью, оптимизация запросов не требует подхода «все или ничего». Производительность можно значительно повысить, если оптимизировать всего несколько часто выполняемых запросов. Рассмотрим ряд способов, как их обнаружить.

Статистика выполнения на основе кэша планов

Как правило, SQL Server кэширует и повторно использует планы выполнения запросов. Для каждого кэшированного плана доступна статистика выполнения,

в том числе количество запусков запроса, совокупное время ЦП и нагрузка ввода/вывода. Эти данные можно использовать, чтобы быстро выявить самые ресурсоемкие запросы, требующие оптимизации. (О кэшировании мы подробнее поговорим в главе 6.)

Анализировать статистику выполнения на основе кэша планов — не самый исчерпывающий метод обнаружения неоптимизированных запросов, и у него немало ограничений. Тем не менее он очень прост в использовании, и им часто удается обойтись. Этот метод работает во всех версиях SQL Server и всегда доступен в системе. Для него не надо настраивать дополнительный мониторинг и специально собирать данные.

Получить статистику выполнения можно с помощью представления `sys.dm_exec_query_stats`[1], как показано в листинге 4.1. Этот запрос немного упрощен, зато он демонстрирует, как работает представление, и позволяет изучить несколько метрик из него. Далее в этой главе я приведу более сложную версию кода, основанную на этом запросе. В зависимости от версии SQL Server и установленных обновлений у вас могут не поддерживаться некоторые столбцы, которые я использую здесь и далее. Если так, то удалите их.

Этот код предоставляет планы выполнения запросов. Есть две функции, которые позволяют их получить:

`sys.dm_exec_query_plan`

Эта функция[2] возвращает план выполнения всего пакета выполнения в формате `XML`. Из-за внутренних ограничений функции размер результирующего `XML` не может превышать 2 Мбайт, а для сложных планов функция может возвращать `NULL`.

`sys.dm_exec_text_query_plan`

Эта функция[3], которая используется в листинге 4.1, возвращает текстовое представление плана выполнения. Его можно получить для всего пакета или для определенной инструкции из пакета, передав смещение инструкции в качестве параметра функции.

В листинге 4.1 планы преобразуются в `XML`-представление с помощью функции `TRY_CONVERT`, которая возвращает `NULL`, если размер `XML` превышает 2 Мбайт. `TRY_CONVERT` можно удалить, если вы работаете с большими планами или запускаете код в SQL Server версий с 2005 по 2008R2.

---

[1] https://oreil.ly/waA0W

[2] https://oreil.ly/xeQEf

[3] https://oreil.ly/utOjM

Листинг 4.1. Использование представления `sys.dm_exec_query_stats`

```
;WITH Queries
AS
(
    SELECT TOP 50
        qs.creation_time AS [Cached Time]
        ,qs.last_execution_time AS [Last Exec Time]
        ,qs.execution_count AS [Exec Cnt]
        ,CONVERT(DECIMAL(10,5),
            IIF
            (
                DATEDIFF(SECOND,qs.creation_time, qs.last_execution_time) = 0
                ,NULL
                ,1.0 * qs.execution_count /
                DATEDIFF(SECOND,qs.creation_time, qs.last_execution_time)
        )
    ) AS [Exec Per Second]
    ,(qs.total_logical_reads + qs.total_logical_writes) /
        qs.execution_count AS [Avg IO]
    ,(qs.total_worker_time / qs.execution_count / 1000)
        AS [Avg CPU(ms)]
    ,qs.total_logical_reads AS [Total Reads]
    ,qs.last_logical_reads AS [Last Reads]
    ,qs.total_logical_writes AS [Total Writes]
    ,qs.last_logical_writes AS [Last Writes]
    ,qs.total_worker_time / 1000 AS [Total Worker Time]
    ,qs.last_worker_time / 1000 AS [Last Worker Time]
    ,qs.total_elapsed_time / 1000 AS [Total Elapsed Time]
    ,qs.last_elapsed_time / 1000 AS [Last Elapsed Time]
    ,qs.total_rows AS [Total Rows]
    ,qs.last_rows AS [Last Rows]
    ,qs.total_rows / qs.execution_count AS [Avg Rows]
    ,qs.total_physical_reads AS [Total Physical Reads]
    ,qs.last_physical_reads AS [Last Physical Reads]
    ,qs.total_physical_reads / qs.execution_count
        AS [Avg Physical Reads]
    ,qs.total_grant_kb AS [Total Grant KB]
    ,qs.last_grant_kb AS [Last Grant KB]
    ,(qs.total_grant_kb / qs.execution_count)
        AS [Avg Grant KB]
    ,qs.total_used_grant_kb AS [Total Used Grant KB]
    ,qs.last_used_grant_kb AS [Last Used Grant KB]
    ,(qs.total_used_grant_kb / qs.execution_count)
        AS [Avg Used Grant KB]
    ,qs.total_ideal_grant_kb AS [Total Ideal Grant KB]
    ,qs.last_ideal_grant_kb AS [Last Ideal Grant KB]
    ,(qs.total_ideal_grant_kb / qs.execution_count)
        AS [Avg Ideal Grant KB]
    ,qs.total_columnstore_segment_reads
        AS [Total CSI Segments Read]
    ,qs.last_columnstore_segment_reads
        AS [Last CSI Segments Read]
    ,(qs.total_columnstore_segment_reads / qs.execution_count)
```

```
        AS [AVG CSI Segments Read]
    ,qs.max_dop AS [Max DOP]
    ,qs.total_spills AS [Total Spills]
    ,qs.last_spills AS [Last Spills]
    ,(qs.total_spills / qs.execution_count) AS [Avg Spills]
    ,qs.statement_start_offset
    ,qs.statement_end_offset
    ,qs.plan_handle
    ,qs.sql_handle
FROM
    sys.dm_exec_query_stats qs WITH (NOLOCK)
ORDER BY
    [Avg IO] DESC
)
SELECT
    SUBSTRING(qt.text, (qs.statement_start_offset/2)+1,
    ((
        CASE qs.statement_end_offset
            WHEN -1 THEN DATALENGTH(qt.text)
            ELSE qs.statement_end_offset
        END - qs.statement_start_offset)/2)+1) AS SQL
    ,TRY_CONVERT(xml,qp.query_plan) AS [Query Plan]
    ,qs.*
FROM
    Queries qs
        OUTER APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
        OUTER APPLY
            sys.dm_exec_text_query_plan
            (
                qs.plan_handle
                ,qs.statement_start_offset
                ,qs.statement_end_offset
            ) qp
OPTION (RECOMPILE, MAXDOP 1);
```

В зависимости от целей настройки можно сортировать данные по-разному: по устройствам ввода/вывода, когда задача — уменьшить нагрузку на диск, по ЦП в системах с привязкой к ЦП и т. д.

На рис. 4.1 частично показан вывод кода на одном из серверов. Как видите, запросы для оптимизации легко отбираются по частоте их выполнения и по данным о потреблении ресурсов.

Планы выполнения, которые отражены в выводе, не содержат фактических показателей выполнения. В этом отношении они аналогичны расчетным планам выполнения. Учитывайте это во время оптимизации (подробнее об этом я расскажу в главе 5).

Этой проблемы можно избежать в SQL Server 2019 и более поздних версиях, а также в базах данных Azure SQL, где можно включить сбор последнего *фактического* плана выполнения инструкций в базах данных с уровнем совместимо-

сти 150. Понадобится также включить параметр базы данных `LAST_QUERY_PLAN_STATS`. Как и в случае любого сбора данных, этот параметр увеличит нагрузку на систему, хотя обычно это увеличение несущественно.

	SQL	Query Plan	Cached Time	Last Exec Time	Exec Cnt
1	Select * in	<ShowPlanXML	2022-02-22 12:01:11.1	2022-02-22 12:01:13.0	1
2	merge bi.or	<ShowPlanXML	2022-02-22 11:51:59.9	2022-02-22 11:52:03.3	1
3	insert into	<ShowPlanXML	2022-02-22 11:05:15.6	2022-02-22 11:05:17.6	1
4	select * in	<ShowPlanXML	2022-02-22 11:02:07.3	2022-02-22 11:02:07.3	1
5	select * in	<ShowPlanXML	2022-02-22 11:03:37.4	2022-02-22 11:03:38.6	1
6	With PreQ a	*NULL*	2022-02-22 11:11:19.6	2022-02-22 11:16:16.7	3
7	UPDATE bi …	<ShowPlanXML	2022-02-20 21:34:13.9	2022-02-22 11:59:48.2	9
8	insert into	<ShowPlanXML	2022-02-19 01:12:05.5	2022-02-22 11:56:58.5	169
9	Merge bi.ac	<ShowPlanXML	2022-02-22 12:30:42.5	2022-02-22 12:30:42.6	1
10	insert into	<ShowPlanXML	2022-02-21 06:19:44.3	2022-02-22 12:31:25.5	64

Exec Per Second	Avg IO	Avg CPU(ms)	Total Reads	Last Reads	Total Writes
0.50000	635122575	4687180	635120267	635120267	2308
0.25000	150500130	237832	150209652	150209652	290478
0.50000	67049164	156828	66824528	66824528	224636
NULL	22608784	35622	22555427	22555427	53357
1.00000	20763067	67591	20719876	20719876	43191
0.01010	20455292	60050	61346943	20456593	18934
0.00007	16741751	56048	150667788	16642890	7972
0.00057	11209013	14492	1891521510	12295322	2801826
NULL	10895279	8734	10868145	10868145	27134
0.00059	6982600	31306	424448642	6609908	22437782

Рис. 4.1. Частичный вывод представления `sys.dm_exec_query_stats`

Последний фактический план выполнения доступен через функцию `sys.dm_exec_query_plan_stats`[1]. Во всех примерах кода в этой главе ее можно использовать вместо `sys.dm_exec_text_query_plan`, и код останется работоспособным.

Следует учитывать еще несколько важных ограничений. Прежде всего вы не увидите никаких данных о запросах, для которых не кэшированы планы выполнения. Могут оказаться пропущенными некоторые редко выполняемые запросы,

[1] https://oreil.ly/Rdf3U

чьих планов уже нет в кэше. Обычно это не проблема, потому что такие запросы чаще всего не нуждаются в оптимизации в начале настройки.

Есть и еще один нюанс. SQL Server не кэширует планы выполнения, если вы используете перекомпиляцию на уровне инструкций с нерегламентированными инструкциями или выполняете хранимые процедуры с предложением `RECOMPILE`. Эти запросы придется захватывать с помощью хранилища запросов или расширенных событий, о которых я расскажу позже в этой главе.

Если вы используете перекомпиляцию на уровне инструкций в хранимых процедурах или других модулях T-SQL, то SQL Server кэширует план выполнения инструкции. Правда, план не будет использоваться повторно, а в статистике выполнения отразится только одно последнее выполнение.

Вторая проблема связана с тем, как долго планы хранятся в кэше. Это зависит от плана, и результаты могут исказиться при сортировке данных по *общим* показателям. Например, запрос с меньшим *средним* временем потребления ЦП может показать большее *общее* количество выполнений и время ЦП, чем запрос с более высоким *средним* временем ЦП, — в зависимости от того, когда каждый из планов был закэширован.

Можно использовать любую из этих метрик, но у каждой свои недостатки. Если сортировать данные по *средним* значениям, то наверху списка могут оказаться редко выполняемые запросы. Например, так происходит с ресурсоемкими задачами по обслуживанию в ночное время. В то же время сортировка по *общим* значениям может пропустить запросы с недавно кэшированными планами.

Можно изучить столбцы `creation_time` и `last_execution_time`, в которых содержится время последнего кэширования и выполнения планов соответственно. Я обычно просматриваю данные, отсортированные как по *общим*, так и по *средним* показателям, учитывая частоту выполнения (общее и среднее количество выполнений в секунду). Я сопоставляю данные из обоих источников, прежде чем решить, что оптимизировать.

Бывает проблема сложнее: для одного и того же запроса или нескольких похожих запросов можно получить несколько результатов. Это может случиться с нерегламентированными рабочими нагрузками, а также когда клиенты используют разные настройки `SET` в своих сеансах, когда пользователи запускают одни и те же запросы с немного различающимся форматированием, и во многих других случаях. Проблема также встречается в базах данных с уровнем совместимости 160 (SQL Server 2022) из-за функции оптимизации плана с учетом параметров (подробнее об этом в главе 6).

К счастью, эту проблему можно преодолеть с помощью столбцов `query_hash` и `query_plan_hash`, которые отображаются в представлении `sys.dm_exec_query_`

stats. Идентичные значения в этих столбцах сигнализируют о похожих запросах и планах выполнения. Эти столбцы можно использовать, чтобы группировать данные.

Оператор DBCC FREEPROCCACHE очищает кэш планов, чтобы уменьшить объем вывода. Не запускайте код из листинга 4.2 на рабочих серверах!

Рассмотрим простой пример. Код из листинга 4.2 выполняет три запроса, а затем изучает содержимое кэша планов. Первые два запроса одинаковы, просто отформатированы по-разному. Третий запрос от них отличается.

Листинг 4.2. Столбцы query_hash и query_plan_hash в действии

```
DBCC FREEPROCCACHE -- Не используйте в промышленной системе!
GO
SELECT /*V1*/ TOP 1 object_id FROM sys.objects WHERE object_id = 1;
GO
SELECT /*V2*/ TOP 1 object_id
FROM sys.objects
WHERE object_id = 1;
GO
SELECT COUNT(*) FROM sys.objects
GO

SELECT
    qs.query_hash, qs.query_plan_hash, qs.sql_handle, qs.plan_handle,
    SUBSTRING(qt.text, (qs.statement_start_offset/2)+1,
    ((
        CASE qs.statement_end_offset
            WHEN -1 THEN DATALENGTH(qt.text)
            ELSE qs.statement_end_offset
        END - qs.statement_start_offset)/2)+1
    ) as SQL
FROM
    sys.dm_exec_query_stats qs
        CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
ORDER BY query_hash
OPTION (MAXDOP 1, RECOMPILE);
```

Результаты показаны на рис. 4.2. В выводимых данных три плана выполнения. В последних двух строках содержатся одинаковые значения query_hash и query_plan_hash, но разные sql_handle и plan_handle.

В листинге 4.3 приведена более сложная версия сценария из листинга 4.1: теперь статистика по похожим запросам группируется. Инструкция и планы выполнения выбираются случайным образом из первого запроса в каждой группе; учитывайте это при анализе.

	query_hash	query_plan_hash	sql_handle
1	0xABDF4B6F62E6CAB0	0xADFF8147759B43B8	0x020000016AE0C0B001D08E62EFCC38CF0A7C26D920AB
2	0xC4EB59564B047EB4	0xF75077CD192BF0C4	0x020000007F0DE40F795F3792B222370C41D02238CAB50
3	0xC4EB59564B047EB4	0xF75077CD192BF0C4	0x02000000E145B236E584443C694C2299F79EE505355D5I

plan_handle	SQL
0x0600010016AE0C0B9010C2DE560200000100000000(	SELECT COUNT(*) FROM sys.objects
0x060001007F0DE40FE00CC2DE560200000100000000(	SELECT /*V2*/ TOP 1 object_id FROM sys.objects
0x06000100E145B2363009C2DE560200000100000000(	SELECT /*V1*/ TOP 1 object_id FROM sys.objects

Рис. 4.2. Несколько планов с одинаковыми значениями `query_hash` и `query_plan_hash`

Листинг 4.3. Использование представления `sys.dm_exec_query_stats` с группировкой по `query_hash`

```
;WITH Data
AS
(
    SELECT TOP 50
        qs.query_hash
        ,COUNT(*) as [Plan Count]
        ,MIN(qs.creation_time) AS [Cached Time]
        ,MAX(qs.last_execution_time) AS [Last Exec Time]
        ,SUM(qs.execution_count) AS [Exec Cnt]
        ,SUM(qs.total_logical_reads) AS [Total Reads]
        ,SUM(qs.total_logical_writes) AS [Total Writes]
        ,SUM(qs.total_worker_time / 1000) AS [Total Worker Time]
        ,SUM(qs.total_elapsed_time / 1000) AS [Total Elapsed Time]
        ,SUM(qs.total_rows) AS [Total Rows]
        ,SUM(qs.total_physical_reads) AS [Total Physical Reads]
        ,SUM(qs.total_grant_kb) AS [Total Grant KB]
        ,SUM(qs.total_used_grant_kb) AS [Total Used Grant KB]
        ,SUM(qs.total_ideal_grant_kb) AS [Total Ideal Grant KB]
        ,SUM(qs.total_columnstore_segment_reads)
            AS [Total CSI Segments Read]
        ,MAX(qs.max_dop) AS [Max DOP]
        ,SUM(qs.total_spills) AS [Total Spills]
    FROM
        sys.dm_exec_query_stats qs WITH (NOLOCK)
    GROUP BY
        qs.query_hash
    ORDER BY
        SUM((qs.total_logical_reads + qs.total_logical_writes) /
            qs.execution_count) DESC
)
SELECT
    d.[Cached Time]
    ,d.[Last Exec Time]
    ,d.[Plan Count]
```

```
    ,sql_plan.SQL
    ,sql_plan.[Query Plan]
    ,d.[Exec Cnt]
    ,CONVERT(DECIMAL(10,5),
        IIF(datediff(second,d.[Cached Time], d.[Last Exec Time]) = 0,
            NULL,
            1.0 * d.[Exec Cnt] /
                datediff(second,d.[Cached Time], d.[Last Exec Time])
        )
    ) AS [Exec Per Second]
    ,(d.[Total Reads] + d.[Total Writes]) / d.[Exec Cnt] AS [Avg IO]
    ,(d.[Total Worker Time] / d.[Exec Cnt] / 1000) AS [Avg CPU(ms)]
    ,d.[Total Reads]
    ,d.[Total Writes]
    ,d.[Total Worker Time]
    ,d.[Total Elapsed Time]
    ,d.[Total Rows]
    ,d.[Total Rows] / d.[Exec Cnt] AS [Avg Rows]
    ,d.[Total Physical Reads]
    ,d.[Total Physical Reads] / d.[Exec Cnt] AS [Avg Physical Reads]
    ,d.[Total Grant KB]
    ,d.[Total Grant KB] / d.[Exec Cnt] AS [Avg Grant KB]
    ,d.[Total Used Grant KB]
    ,d.[Total Used Grant KB] / d.[Exec Cnt] AS [Avg Used Grant KB]
    ,d.[Total Ideal Grant KB]
    ,d.[Total Ideal Grant KB] / d.[Exec Cnt] AS [Avg Ideal Grant KB]
    ,d.[Total CSI Segments Read]
    ,d.[Total CSI Segments Read] / d.[Exec Cnt] AS [AVG CSI Segments Read]
    ,d.[Max DOP]
    ,d.[Total Spills]
    ,d.[Total Spills] / d.[Exec Cnt] AS [Avg Spills]
FROM
    Data d
        CROSS APPLY
        (
            SELECT TOP 1
                SUBSTRING(qt.text, (qs.statement_start_offset/2)+1,
                ((
                     CASE qs.statement_end_offset
                         WHEN -1 THEN DATALENGTH(qt.text)
                         ELSE qs.statement_end_offset
                     END – qs.statement_start_offset)/2)+1
                 ) AS SQL
                 ,TRY_CONVERT(XML,qp.query_plan) AS [Query Plan]
             FROM
                 sys.dm_exec_query_stats qs
                     OUTER APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
                     OUTER APPLY sys.dm_exec_text_query_plan
                     (
                          qs.plan_handle
                          ,qs.statement_start_offset
                          ,qs.statement_end_offset
```

```
                    ) qp
            WHERE
                qs.query_hash = d.query_hash AND ISNULL(qt.text,'') <> ''
        ) sql_plan
ORDER BY
    [Avg IO] DESC
OPTION (RECOMPILE, MAXDOP 1);
```

Начиная с SQL Server 2008, можно получать статистику выполнения хранимых процедур с помощью представления `sys.dm_exec_procedure_stats`[1]. Также можно использовать код из листинга 4.4. Как и в представлении `sys.dm_exec_query_stats`, данные можно сортировать по разным показателям выполнения в зависимости от вашей стратегии оптимизации. Учтите, что в статистику выполнения входят метрики из динамического SQL и других вложенных модулей (хранимых процедур, функций, триггеров), вызываемых из хранимых процедур.

Листинг 4.4. Использование представления `sys.dm_exec_procedure_stats`

```
SELECT TOP 50
    IIF (ps.database_id = 32767,
        'mssqlsystemresource',
        DB_NAME(ps.database_id)
    ) AS [DB]
    ,OBJECT_NAME(
        ps.object_id,
        IIF(ps.database_id = 32767, 1, ps.database_id)
    ) AS [Proc Name]
    ,ps.type_desc AS [Type]
    ,ps.cached_time AS [Cached Time]
    ,ps.last_execution_time AS [Last Exec Time]
    ,qp.query_plan AS [Plan]
    ,ps.execution_count AS [Exec Count]
    ,CONVERT(DECIMAL(10,5),
        IIF(datediff(second,ps.cached_time, ps.last_execution_time) = 0,
            NULL,
            1.0 * ps.execution_count /
                datediff(second,ps.cached_time, ps.last_execution_time)
        )
    ) AS [Exec Per Second]
    ,(ps.total_logical_reads + ps.total_logical_writes) /
        ps.execution_count AS [Avg IO]
    ,(ps.total_worker_time / ps.execution_count / 1000)
        AS [Avg CPU(ms)]
    ,ps.total_logical_reads AS [Total Reads]
    ,ps.last_logical_reads AS [Last Reads]
    ,ps.total_logical_writes AS [Total Writes]
    ,ps.last_logical_writes AS [Last Writes]
    ,ps.total_worker_time / 1000 AS [Total Worker Time]
```

[1] https://oreil.ly/B2Xyg

```
    ,ps.last_worker_time / 1000 AS [Last Worker Time]
    ,ps.total_elapsed_time / 1000 AS [Total Elapsed Time]
    ,ps.last_elapsed_time / 1000 AS [Last Elapsed Time]
    ,ps.total_physical_reads AS [Total Physical Reads]
    ,ps.last_physical_reads AS [Last Physical Reads]
    ,ps.total_physical_reads / ps.execution_count AS [Avg Physical Reads]
    ,ps.total_spills AS [Total Spills]
    ,ps.last_spills AS [Last Spills]
    ,(ps.total_spills / ps.execution_count) AS [Avg Spills]
FROM
    sys.dm_exec_procedure_stats ps WITH (NOLOCK)
        CROSS APPLY sys.dm_exec_query_plan(ps.plan_handle) qp
ORDER BY
    [Avg IO] DESC
OPTION (RECOMPILE, MAXDOP 1);
```

На рис. 4.3 частично показан вывод этого кода. Как видите, таким образом можно получить планы выполнения хранимых процедур. Во внутреннем представлении планы выполнения хранимых процедур и других модулей T-SQL представляют собой просто наборы отдельных планов каждой инструкции. В некоторых случаях — например, когда размер плана выполнения превышает 2 Мбайт — план не попадает в выходные данные.

	DB	Proc Name	Type	Cached Time	Last Exec Time
1		IndexOptimize	SQL_STORED_PROCEDURE	2021-01-03 01:05:01.067	2021-01-10 01:05:00.640
2		archive_data	SQL_STORED_PROCEDURE	2021-01-10 05:45:00.503	2021-01-12 05:45:00.367
3		archive_misc	SQL_STORED_PROCEDURE	2021-01-03 05:47:42.547	2021-01-12 05:49:36.920
4		AGGREGATE_LI…	SQL_STORED_PROCEDURE	2021-01-10 03:01:48.407	2021-01-12 18:19:46.477
5		archive_orde…	SQL_STORED_PROCEDURE	2021-01-10 05:45:03.197	2021-01-12 05:45:06.397
6		archive_inte…	SQL_STORED_PROCEDURE	2021-01-11 05:47:20.680	2021-01-12 05:48:32.947

Plan	Exec Count	Exec Per Second	Avg IO	Avg CPU(ms)
NULL	2	0.00000	997605286	3231184
<ShowPlanXML xmlns="http:/…	3	0.00002	177832619	940548
<ShowPlanXML xmlns="http:/…	10	0.00001	80059850	677919
<ShowPlanXML xmlns="http:/…	404	0.00177	44012127	117734
NULL	3	0.00002	43522659	73929
<ShowPlanXML xmlns="http:/…	2	0.00002	23452696	40479

Рис. 4.3. Частичный вывод представления `sys.dm_exec_procedure_stats`

Листинг 4.5 помогает решить эту проблему. С помощью этого кода можно получить кэшированные планы выполнения и их метрики для отдельных инструкций из модулей T-SQL. Имя модуля указывается в предложении `WHERE` при запуске скрипта.

Листинг 4.5. Получение плана выполнения и статистики для операторов хранимой процедуры

```
SELECT
    qs.creation_time AS [Cached Time]
    ,qs.last_execution_time AS [Last Exec Time]
    ,SUBSTRING(qt.text, (qs.statement_start_offset/2)+1,
    ((
        CASE qs.statement_end_offset
            WHEN -1 THEN DATALENGTH(qt.text)
            ELSE qs.statement_end_offset
        END – qs.statement_start_offset)/2)+1) AS SQL
    ,TRY_CONVERT(XML,qp.query_plan) AS [Query Plan]
    ,CONVERT(DECIMAL(10,5),
        IIF(datediff(second,qs.creation_time, qs.last_execution_time) = 0,
            NULL,
            1.0 * qs.execution_count /
                datediff(second,qs.creation_time, qs.last_execution_time)
        )
    ) AS [Exec Per Second]
    ,(qs.total_logical_reads + qs.total_logical_writes) /
        qs.execution_count AS [Avg IO]
    ,(qs.total_worker_time / qs.execution_count / 1000)
        AS [Avg CPU(ms)]
    ,qs.total_logical_reads AS [Total Reads]
    ,qs.last_logical_reads AS [Last Reads]
    ,qs.total_logical_writes AS [Total Writes]
    ,qs.last_logical_writes AS [Last Writes]
    ,qs.total_worker_time / 1000 AS [Total Worker Time]
    ,qs.last_worker_time / 1000 AS [Last Worker Time]
    ,qs.total_elapsed_time / 1000 AS [Total Elapsed Time]
    ,qs.last_elapsed_time / 1000 AS [Last Elapsed Time]
    ,qs.total_rows AS [Total Rows]
    ,qs.last_rows AS [Last Rows]
    ,qs.total_rows / qs.execution_count AS [Avg Rows]
    ,qs.total_physical_reads AS [Total Physical Reads]
    ,qs.last_physical_reads AS [Last Physical Reads]
    ,qs.total_physical_reads / qs.execution_count
        AS [Avg Physical Reads]
    ,qs.total_grant_kb AS [Total Grant KB]
    ,qs.last_grant_kb AS [Last Grant KB]
    ,(qs.total_grant_kb / qs.execution_count)
        AS [Avg Grant KB]
    ,qs.total_used_grant_kb AS [Total Used Grant KB]
    ,qs.last_used_grant_kb AS [Last Used Grant KB]
    ,(qs.total_used_grant_kb / qs.execution_count)
        AS [Avg Used Grant KB]
    ,qs.total_ideal_grant_kb AS [Total Ideal Grant KB]
    ,qs.last_ideal_grant_kb AS [Last Ideal Grant KB]
    ,(qs.total_ideal_grant_kb / qs.execution_count)
        AS [Avg Ideal Grant KB]
    ,qs.total_columnstore_segment_reads
```

```
        AS [Total CSI Segments Read]
    ,qs.last_columnstore_segment_reads
        AS [Last CSI Segments Read]
    ,(qs.total_columnstore_segment_reads / qs.execution_count)
        AS [AVG CSI Segments Read]
    ,qs.max_dop AS [Max DOP]
    ,qs.total_spills AS [Total Spills]
    ,qs.last_spills AS [Last Spills]
    ,(qs.total_spills / qs.execution_count) AS [Avg Spills]
FROM
    sys.dm_exec_query_stats qs WITH (NOLOCK)
        OUTER APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
        OUTER APPLY sys.dm_exec_text_query_plan
        (
            qs.plan_handle
            ,qs.statement_start_offset
            ,qs.statement_end_offset
        ) qp
WHERE
    OBJECT_NAME(qt.objectid, qt.dbid) = <SP Name>
                                        -- Замените на имя хранимой процедуры
ORDER BY
    qs.statement_start_offset, qs.statement_end_offset
OPTION (RECOMPILE, MAXDOP 1);
```

Начиная с SQL Server 2016, можно получать статистику выполнения для триггеров и скалярных пользовательских функций, используя представления `sys.dm_exec_trigger_stats`[1] и `sys.dm_exec_function_stats`[2] соответственно. Можно использовать тот же код, что и в листинге 4.4, просто заменив в нем имя динамического административного представления. Код также можно скачать из сопроводительных материалов этой книги.

Наконец, стоит отметить, что SQL Server может кэшировать тысячи планов выполнения. Кроме того, функции получения планов запросов и инструкций SQL весьма трудоемкие, поэтому я использую параметр запроса `MAXDOP 1`, чтобы уменьшить накладные расходы. В некоторых случаях стоит сохранять содержимое кэша планов в отдельную базу данных с помощью операторов `SELECT INTO` и анализировать данные на тестовых серверах.

У статистики выполнения на основе кэша планов есть определенные ограничения, и некоторые запросы могут быть пропущены. Но этот подход годится в качестве отправной точки. Важнее всего, что нужные данные собираются автоматически и доступны сразу без дополнительных инструментов мониторинга.

---

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

[2] https://oreil.ly/mZEJO

Расширенные события и трассировки SQL

Уверен, что каждый специалист по SQL Server знает о трассировках SQL (SQL Traces) и расширенных событиях (Extended Events или xEvents). Они позволяют захватывать различные события в системе для анализа и устранения неполадок в режиме реального времени. С их помощью также можно «отлавливать» длительные и ресурсоемкие запросы, в том числе те, которые не кэшируют планы выполнения и поэтому не попадают в представление `sys.dm_exec_query_stats`.

Тем не менее мне хотелось бы начать этот раздел с предупреждения: *не используйте* для захвата запросов трассировки SQL и расширенные события без *крайней необходимости*. Захватывать выполняющиеся инструкции — это ресурсоемкая операция, которая может значительно ухудшить производительность в высоконагруженных системах. (В главе 1 мы уже видели такой пример.)

Неважно, сколько данных вы хотите собрать. Бóльшую часть инструкций можно исключить из вывода, то есть отфильтровать запросы с низким потреблением ресурсов. Но SQL Server все равно вынужден захватывать все инструкции, чтобы вычислять их, фильтровать и отбрасывать лишние события.

Не собирайте ненужную информацию для событий, которые вы анализируете, или для захватываемых действий расширенных событий. Некоторые действия, например `callstack`, обходятся очень дорого и существенно сказываются на производительности системы. По возможности используйте расширенные события вместо трассировки SQL. Они легче и влекут за собой меньше накладных расходов.

В табл. 4.1 показано несколько расширенных событий и событий трассировки SQL, с помощью которых можно обнаруживать неэффективные запросы. Каждому из этих событий, кроме `sqlserver.attention`, соответствует некоторое другое событие, которое срабатывает в начале исполнения. Иногда эти события тоже нужно захватывать, чтобы сопоставить рабочие нагрузки из нескольких сеансов.

Какие события нужно захватывать? Это зависит от рабочей нагрузки системы, структуры уровня доступа к данным и вашей стратегии устранения неполадок. Например, события `sqlserver.sql_statement_completed` и `sqlserver.sp_statement_completed` позволяют обнаруживать неэффективные нерегламентированные запросы и запросы модулей T-SQL. А неэффективные пакеты и хранимые процедуры можно выловить с помощью событий `sqlserver.sql_batch_completed` и `sqlserver.rpc_completed`, не создавая чрезмерной дополнительной нагрузки на систему.

Таблица 4.1. Расширенные события и события трассировки SQL для обнаружения неэффективных запросов

Расширенное событие	Событие трассировки SQL	Описание
`sqlserver.sql_statement_completed`	`SQL:StmtCompleted`	Возникает, когда заканчивается выполнение инструкции
`sqlserver.sp_statement_completed`	`SP:StmtCompleted`	Возникает, когда заканчивается выполнение инструкции SQL в модуле T-SQL
`sqlserver.rpc_completed`	`RPC:Completed`	Возникает, когда заканчивается выполнение удаленного вызова процедуры (RPC). RPC — это параметризованные SQL-запросы, такие как вызовы хранимых процедур или параметризованные пакеты, направленные из сторонних приложений. Многие клиентские библиотеки запускают запросы через функцию sp_executesql, и ее вызовы захватываются этим событием
`sqlserver.module_end`	`SP:Completed`	Возникает, когда заканчивается выполнение модуля T-SQL
`sqlserver.sql_batch_completed`	`SQL:BatchCompleted`	Возникает, когда заканчивается выполнение пакета SQL
`sqlserver.attention`	`Error:Attention`	Возникает, когда клиент отменяет выполнение запроса по истечении времени ожидания или явным образом (например, с помощью красной кнопки отмены в SSMS)

Таким же образом решайте, какие расширенные действия захватывать. Например, можно игнорировать информацию о пользователе и клиентском приложении, если она не нужна для устранения неполадок. В то же время можно собирать действия `query_hash` и `query_plan_hash` и использовать их, чтобы оценивать совокупное влияние похожих запросов и планов выполнения.

Обычно я диагностирую неэффективные запросы в двух ситуациях. Во-первых, я могу запустить сеанс на несколько минут, записывая результаты в целевой объект `ring_buffer`. Обычно я так поступаю, когда нагрузка в системе относительно статична и ее можно репрезентативно отразить простой выборкой. Во-вторых, я могу запустить расширенный сеанс на несколько часов, используя `event_file` в качестве целевого объекта.

В листинге 4.6 показан второй способ, который сохраняет данные в папку *C:\ExtEvents* (в вашей системе путь может быть другим). Этот код захватывает инструкции, которые потребляют более 5000 мс процессорного времени или производят более 50 000 логических операций чтения или записи. Код в листингах 4.6 и 4.7 будет работать в SQL Server 2012 и более поздних версиях, однако его придется модифицировать для SQL Server 2008, который по-другому работает с целевым файлом и не поддерживает действия `query_hash` и `query_plan_hash`.

Должен предупредить, что этот код добавляет накладные расходы, масштаб которых зависит от рабочей нагрузки и объема собираемых данных. Не оставляйте этот сеанс активным, когда не занимаетесь устранением неполадок производительности. Кроме того, подберите пороговые значения `cpu_time`, `logical_reads` и `writes` в соответствии со своей рабочей нагрузкой и не захватывайте слишком много запросов.

Похожим образом составьте список действий расширенных событий на основе вашей стратегии устранения неполадок. К примеру, нет необходимости собирать `plan_handle`, если вы планируете выполнять анализ на другом сервере и не сможете получить планы выполнения из кэша.

Листинг 4.6. Захват запросов с интенсивным использованием ЦП и операций ввода/вывода

```
CREATE EVENT SESSION [Expensive Queries]
ON SERVER
ADD EVENT sqlserver.sql_statement_completed
(
    ACTION
    (
        sqlserver.client_app_name
        ,sqlserver.client_hostname
        ,sqlserver.database_id
        ,sqlserver.plan_handle
        ,sqlserver.query_hash
        ,sqlserver.query_plan_hash
        ,sqlserver.sql_text
        ,sqlserver.username
    )
    WHERE
    (
        (
            cpu_time >= 5000000 or -- Время в микросекундах
            logical_reads >= 50000 or
            writes >= 50000
        ) AND
        sqlserver.is_system = 0
    )
)
,ADD EVENT sqlserver.sp_statement_completed
(
```

```
    ACTION
    (
        sqlserver.client_app_name
        ,sqlserver.client_hostname
        ,sqlserver.database_id
        ,sqlserver.plan_handle
        ,sqlserver.query_hash
        ,sqlserver.query_plan_hash
        ,sqlserver.sql_text
        ,sqlserver.username
    )
    WHERE
    (
        (
            cpu_time >= 5000000 or -- Время в микросекундах
            logical_reads >= 50000 or
            writes >= 50000
        ) AND
        sqlserver.is_system = 0
    )
)
ADD TARGET package0.event_file
(
    SET FILENAME = 'C:\ExtEvents\Expensive Queries.xel'
)
WITH
(
    event_retention_mode=allow_single_event_loss
    ,max_dispatch_latency=30 seconds
);
```

В листинге 4.7 приведен код для анализа собранных данных. Сперва он загружает собранные события во временную таблицу, используя функцию `sys.fn_xe_file_target_read_file`[1]. Звездочка в конце имени файла заставляет SQL Server загрузить все файлы продолжения (rollover files) из сеанса расширенного события.

Затем код анализирует собранные события, сохраняя результаты в другую временную таблицу. Возможно, вам придется изменить код в обобщенном табличном выражении `EventInfo` в зависимости от полей расширенных событий и действий, которые вас интересуют. Не анализируйте неважную информацию, ведь разбирать XML — затратная и долгая процедура.

Наконец, если вы запускаете этот код в SQL Server 2016 или более ранних версиях, его нужно модифицировать, чтобы получать время события из столбца `event_data`, содержащего данные в формате XML. В SQL Server 2017 функция `sys.fn_xe_file_target_read_file` возвращает его сама.

[1] https://oreil.ly/L6sG8

Листинг 4.7. Анализ собранных данных расширенного события

```
CREATE TABLE #EventData
(
  event_data XML NOT NULL,
  file_name NVARCHAR(260) NOT NULL,
  file_offset BIGINT NOT NULL,
  timestamp_utc datetime2(7) NOT NULL -- SQL Server 2017+
);

INSERT INTO #EventData(event_data, file_name, file_offset, timestamp_utc)
  SELECT CONVERT(XML,event_data), file_name, file_offset, timestamp_utc
  FROM sys.fn_xe_file_target_read_file
    ('c:\extevents\Expensive Queries*.xel',NULL,NULL,NULL);

;WITH EventInfo([Event],[Event Time],[DB],[Statement],[SQL],[User Name]
  ,[Client],[App],[CPU Time],[Duration],[Logical Reads]
  ,[Physical Reads],[Writes],[Rows],[Query Hash],[Plan Hash]
  ,[PlanHandle],[Stmt Offset],[Stmt Offset End],File_Name,File_Offset)
AS
(
  SELECT
    event_data.value('/event[1]/@name','SYSNAME') AS [Event]
    ,timestamp_utc AS [Event Time] -- SQL Server 2017+
    /*,event_data.value('/event[1]/@timestamp','DATETIME')
        AS [Event Time] -- Версии до SQL Server 2017 */
    ,event_data.value
      ('(//event[1]/action[@name="database_id"]/value/text())[1])'
        ,'INT') AS [DB]
    ,event_data.value
      ('(//event[1]/data[@name="statement"]/value/text())[1])'
        ,'NVARCHAR(MAX)') AS [Statement]
    ,event_data.value
      ('(//event[1]/action[@name="sql_text"]/value/text())[1])'
        ,'NVARCHAR(MAX)') AS [SQL]
    ,event_data.value
      ('(//event[1]/action[@name="username"]/value/text())[1])'
        ,'NVARCHAR(255)') AS [User Name]
    ,event_data.value
      ('(//event[1]/action[@name="client_hostname"]/value/text())[1])'
        ,'NVARCHAR(255)') AS [Client]
    ,event_data.value
      ('(//event[1]/action[@name="client_app_name"]/value/text())[1])'
        ,'NVARCHAR(255)') AS [App]
    ,event_data.value
      ('(//event[1]/data[@name="cpu_time"]/value/text())[1])'
        ,'BIGINT') AS [CPU Time]
    ,event_data.value
      ('(//event[1]/data[@name="duration"]/value/text())[1])'
        ,'BIGINT') AS [Duration]
    ,event_data.value
      ('(//event[1]/data[@name="logical_reads"]/value/text())[1])'
        ,'INT') AS [Logical Reads]
```

```
    ,event_data.value
      ('(((/event[1]/data[@name="physical_reads"]/value/text())[1])'
        ,'INT') AS [Physical Reads]
    ,event_data.value
      ('(((/event[1]/data[@name="writes"]/value/text())[1])'
        ,'INT') AS [Writes]
    ,event_data.value
      ('(((/event[1]/data[@name="row_count"]/value/text())[1])'
        ,'INT') AS [Rows]
    ,event_data.value(
      'xs:hexBinary((((/event[1]/action[@name="query_hash"]/value/text())[1]))'
        ,'BINARY(8)') AS [Query Hash]
    ,event_data.value(
      'xs:hexBinary((((/event[1]/action[@name="query_plan_hash"]/value/text())[1]))'
        ,'BINARY(8)') AS [Plan Hash]
    ,event_data.value(
      'xs:hexBinary((((/event[1]/action[@name="plan_handle"]/value/text())[1]))'
        ,'VARBINARY(64)') AS [PlanHandle]
    ,event_data.value
      ('(((/event[1]/data[@name="offset"]/value/text())[1])'
        ,'INT') AS [Stmt Offset]
    ,event_data.value
      ('(((/event[1]/data[@name="offset_end"]/value/text())[1])'
        ,'INT') AS [Stmt Offset End]
    ,file_name
    ,file_offset
  FROM
    #EventData
)
SELECT
  ei.*
  ,TRY_CONVERT(XML,qp.Query_Plan) AS [Plan]
INTO #Queries
FROM
  EventInfo ei
    OUTER APPLY
      sys.dm_exec_text_query_plan
      (
        ei.PlanHandle
        ,ei.[Stmt Offset]
        ,ei.[Stmt Offset End]
      ) qp
OPTION (MAXDOP 1, RECOMPILE);
```

Теперь можно работать с сырыми данными из таблицы `#Queries` и находить самые неэффективные запросы, требующие оптимизации. Часто также полезно группировать данные на основе оператора, хеша запроса или хеша плана, анализируя совокупное влияние запросов.

В сопутствующих материалах этой книги приведен сценарий, с помощью которого можно захватывать рабочую нагрузку в целевой объект `ring_buffer`. Но тут

есть особенность: представление `sys.dm_xe_session_targets`, которое отображает данные, собранные от целевого объекта, способно выводить только 4 Мбайт XML-данных. В результате вы можете просто не увидеть некоторые события.

Повторюсь: *остерегайтесь накладных расходов, которые возникают из-за расширенных событий и трассировки SQL*. Не надо постоянно создавать или запускать их сеансы. Часто можно получить достаточно данных для устранения неполадок, если запустить сеанс всего на несколько минут.

Хранилище запросов

Мы обсудили уже два подхода к обнаружению неэффективных запросов, и у обоих есть ограничения. Статистика на основе кэша планов может пропускать некоторые запросы; трассировки SQL и расширенные события требуют сложного анализа выходных данных и способны сильно ухудшить производительность в высоконагруженных системах.

Хранилище запросов, представленное в SQL Server 2016, помогает избавиться от этих ограничений. Оно похоже на «черный ящик» в самолете. Когда хранилище запросов включено, SQL Server собирает и сохраняет статистику времени выполнения и планы выполнения запросов. В результате видно, как работают планы выполнения и как они меняются со временем. Наконец, при этом можно принудительно задавать конкретные планы выполнения для запросов. Это уменьшает проблемы сканирования параметров, о которых мы еще поговорим в главе 6.

В локальной версии SQL Server вплоть до SQL Server 2019 хранилище запросов по умолчанию отключено. Зато оно по умолчанию включено в базах данных и управляемых экземплярах Azure SQL и новых базах данных, созданных в SQL Server 2022.

Хранилище запросов полностью интегрировано в конвейер обработки запросов, как показано на рис. 4.4.

Когда нужно выполнить запрос, SQL Server ищет план выполнения в кэше планов. Если план найден, SQL Server проверяет, нужно ли перекомпилировать запрос (из-за обновлений статистики или других факторов), был ли создан новый принудительный план и удален ли старый принудительный план из хранилища.

Во время компиляции SQL Server проверяет, есть ли для запроса принудительный план. Если да, то запрос фактически компилируется с принудительным планом, как и в случае указания `USE PLAN`. Если результирующий план действителен (*valid*), он сохраняется в кэше планов для повторного использования.

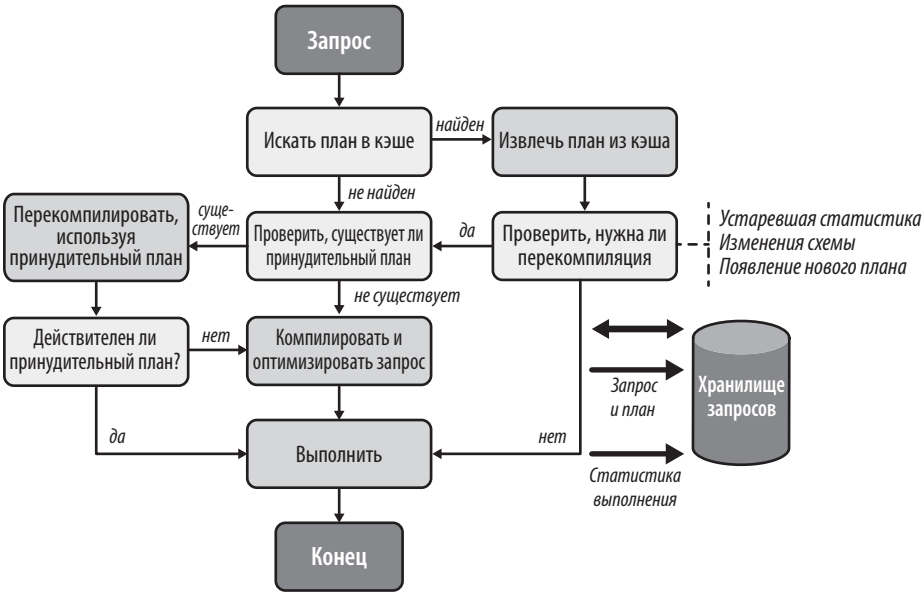


Рис. 4.4. Конвейер обработки запросов

Когда принудительный план больше недействителен (например, если пользователь удалил индекс, на который ссылается план), SQL Server все равно выполнит запрос, однако при компиляции не будет создавать и кэшировать принудительный план. При этом в хранилище запросов останутся оба плана, но принудительный будет помечен как недействительный. Все это происходит прозрачно для приложений.

В SQL Server 2022 и базах данных Azure SQL хранилище запросов позволяет добавлять указания на уровне запроса с помощью хранимой процедуры `sp_query_store_set_hints`[1]. С указаниями хранилища запросов SQL Server будет компилировать и выполнять запросы так же, как если бы вы добавляли указания в предложении `OPTION` в запросе. Это дает больше простора для маневра при настройке запросов и позволяет не модифицировать приложения.

Несмотря на тесную интеграцию с конвейером обработки запросов и внутренние факторы оптимизации, хранилище запросов все равно увеличивает нагрузку на систему. Масштаб этого увеличения зависит от двух основных факторов:

Количество компиляций в системе

Чем больше компиляций выполняет SQL Server, тем больше нагрузки ложится на хранилище запросов. Из-за этого оно может не лучшим образом

[1] https://oreil.ly/xI3nA

работать в системах с очень большой, нерегламентированной, непараметризованной рабочей нагрузкой.

Настройки сбора данных

В настройках хранилища запросов можно указать, нужно ли собирать все запросы или только ресурсоемкие, а также задать интервалы группировки и параметры резервирования данных. Чем больше данных вы собираете и/или чем меньше интервалы группировки, тем больше накладные расходы.

Обратите особое внимание на параметр `QUERY_CAPTURE_MODE`: он определяет, какие запросы захватываются. Если `QUERY_CAPTURE_MODE=ALL` (по умолчанию в SQL Server 2016 и 2017), то хранилище запросов захватывает все запросы в системе. Это сказывается на производительности, особенно при нерегламентированной рабочей нагрузке.

В случае `QUERY_CAPTURE_MODE=AUTO` (по умолчанию в SQL Server 2019 и более поздних версиях) хранилище запросов не захватывает небольшие или редко выполняемые запросы. Обычно этот вариант лучше.

Наконец, начиная с SQL Server 2019, можно установить `QUERY_CAPTURE_MODE=CUSTOM` и вручную настроить критерии захвата запросов.

При грамотной настройке накладные расходы, связанные с хранилищем запросов, будут относительно невелики. Но иногда они оказываются значительными. Например, я использовал хранилище запросов, когда оптимизировал производительность одного процесса, состоящего из очень большого количества небольших нерегламентированных запросов. Я перехватил все запросы в системе в режиме `QUERY_CAPTURE_MODE=ALL`, собрав почти 10 Гбайт данных в хранилище запросов. С включенным хранилищем процесс выполнялся 8 часов, а без него — всего 2,5 часа.

Но если ваша система способна немного поднапрячься, я рекомендую включить хранилище запросов. С ним, в частности, эффективнее работают некоторые функции интеллектуальной обработки запросов (IQP). Также хранилище упрощает настройку запросов и может сэкономить вам много часов работы, если оно включено.

Отслеживайте ожидания `QDS*`, когда включено хранилище запросов. Слишком большие ожидания `QDS*` могут сигнализировать о том, что хранилище создает высокую нагрузку. Ожидания `QDS_PERSIST_TASK_MAIN_LOOP_SLEEP` и `QDS_ASYNC_QUEUE` можно игнорировать.

Есть два важных флага трассировки, которые стоит включить, если используется хранилище запросов:

`T7745`

Чтобы сократить накладные расходы, SQL Server периодически кэширует некоторые данные из хранилища запросов в памяти, сбрасывая их в базу данных. Интервал сброса управляется параметром `DATA_FLUSH_INTERVAL_SECONDS`, от которого зависит, сколько данных из хранилища запросов вы рискуете потерять в случае сбоя SQL Server. Впрочем, обычно SQL Server сбрасывает данные хранилища запросов из памяти во время завершения работы SQL Server или аварийного переключения.

Если включен флаг трассировки `T7745`, SQL Server не сбрасывает данные на диск при выключении и аварийном переключении, что позволяет сэкономить время этих операций в высоконагруженных системах. Потеря небольшого количества данных телеметрии обычно допустима.

`T7752` (SQL Server 2016 и 2017)

При запуске базы данных SQL Server загружает некоторые данные хранилища запросов в память, пока сама база еще недоступна. Если хранилища запросов имеют большой объем, это может увеличить время перезапуска или аварийного переключения SQL Server и затруднить работу пользователей.

Флаг трассировки `T7752` заставляет SQL Server асинхронно загружать данные хранилища, позволяя параллельно с этим выполнять запросы. Во время загрузки не будет собираться телеметрия, но обычно это приемлемо ради более быстрого запуска.

Можно проанализировать влияние синхронной загрузки хранилища запросов, просмотрев время ожидания для типа ожидания `QDS_LOADDB`. Это ожидание происходит только при запуске базы данных, поэтому, чтобы получить нужное значение, надо запросить представление `sys.dm_os_wait_stats` и отфильтровать вывод по типу ожидания.

Как правило, не следует создавать слишком большое хранилище запросов. Кроме того, рекомендую отслеживать размер хранилища, особенно в высоконагруженных системах. Иногда SQL Server не успевает очищать данные достаточно быстро, особенно если вы используете режим сбора `QUERY_CAPTURE_MODE=ALL`.

Наконец, лучше установить последние обновления SQL Server, особенно если вы используете SQL Server 2016 и 2017. С тех пор как хранилище запросов впервые появилось в SQL Server, для него было выпущено много обновлений, которые улучшали масштабируемость и исправляли ошибки.

Работать с хранилищем запросов можно двумя способами: через графический интерфейс в SSMS или напрямую запрашивая динамические административные представления. Давайте сначала посмотрим на графический интерфейс.

Отчеты хранилища запросов в SSMS

Если в базе данных включено хранилище запросов, вы увидите папку *Query Store* на панели *Object Explorer* (рис. 4.5). Количество отчетов в папке будет зависеть от версий SQL Server и SSMS в вашей системе. В оставшейся части этого раздела вы познакомитесь с семью отчетами, показанными на рис. 4.5.

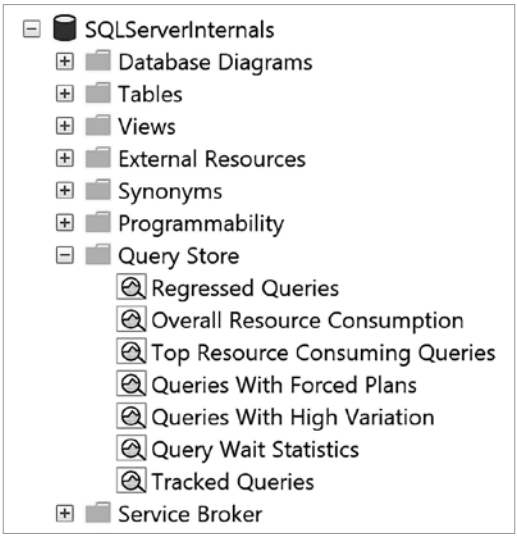


Рис. 4.5. Отчеты хранилища запросов в SSMS

Регрессивные запросы

В отчете, показанном на рис. 4.6, отображаются запросы, производительность которых со временем упала. Для анализа можно настроить временны́е рамки и критерии регрессии: например, дисковые операции, потребление ЦП и количество выполнений.

Выберите запрос на графике в верхней левой части отчета. В верхней правой части отчета перечислены собранные планы выполнения для этого запроса. Можно нажимать на точки, соответствующие различным планам выполнения, и просматривать сами планы внизу. Также можно сравнивать планы выполнения между собой.

Кнопка *Force Plan* позволяет принудительно назначить план запросу. На внутреннем уровне она вызывает хранимую процедуру `sys.sp_query_store_force_plan`[1]. Аналогично кнопка *Unforce Plan* отменяет принудительный план, вызывая хранимую процедуру `sys.sp_query_store_unforce_plan`[2].

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

[2] https://oreil.ly/wa4i9

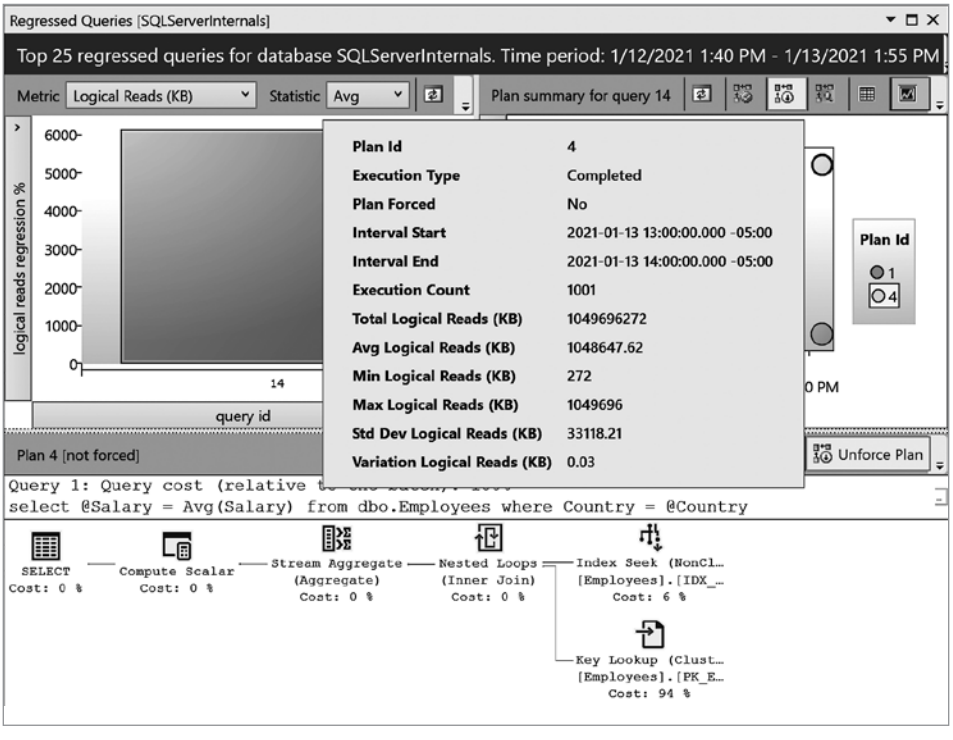


Рис. 4.6. Отчет Regressed Queries

Отчет Regressed Queries — отличный инструмент для устранения неполадок, связанных со сканированием параметров, которое мы обсудим в главе 6. Эти проблемы легко исправить, принудительно задав конкретные планы выполнения.

Наиболее ресурсоемкие запросы

Этот отчет (рис. 4.7) позволяет обнаружить самые ресурсоемкие запросы в системе. Данные в нем аналогичны представлению `sys.dm_exec_query_stats`, однако не зависят от кэша планов. Здесь можно настроить метрики, используемые для сортировки данных, а также временной интервал.

Общее потребление ресурсов

Этот отчет содержит статистику рабочей нагрузки и использования ресурсов за указанные интервалы времени. Он позволяет обнаруживать и анализировать всплески использования ресурсов и детализировать запросы, вызывающие такие всплески. На рис. 4.8 показан вывод отчета.

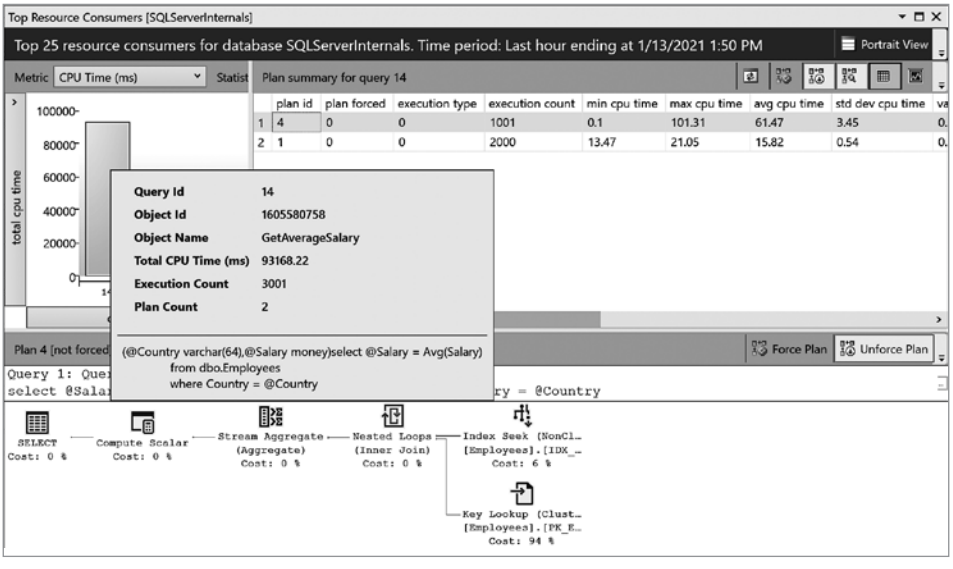


Рис. 4.7. Отчет Resource Consuming Queries

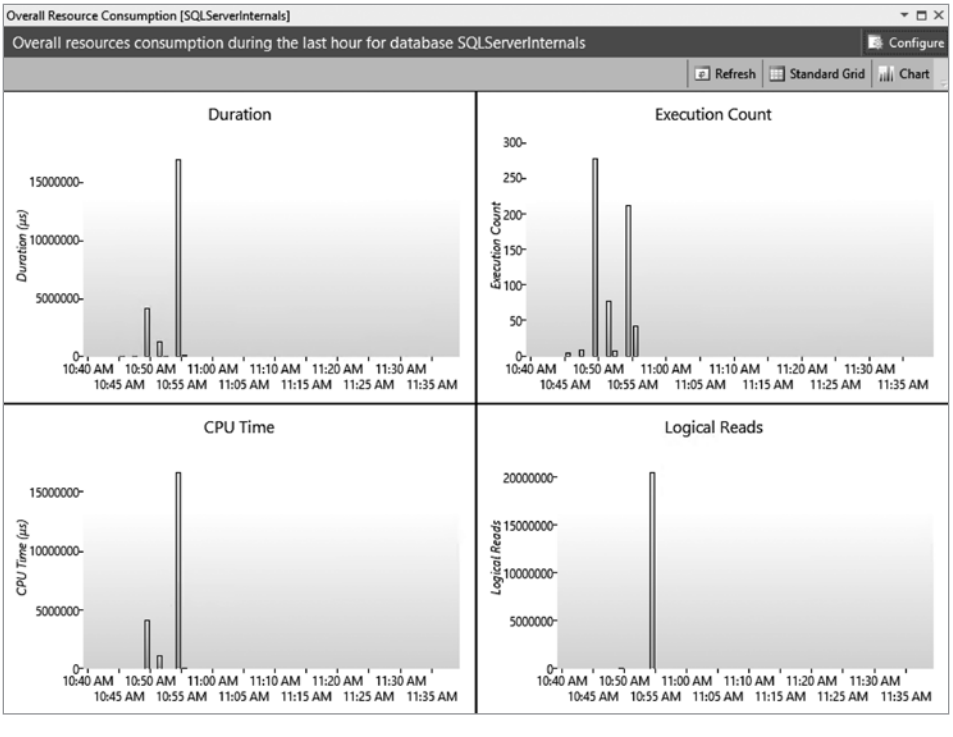


Рис. 4.8. Отчет Overall Resource Consumption

Запросы с высокой вариативностью

Отчет Queries With High Variation позволяет выявить запросы с большими колебаниями производительности. С его помощью можно обнаруживать аномалии в рабочей нагрузке, а также потенциальные очаги снижения производительности. Чтобы сэкономить место в книге, я не привожу снимков этого и последующих отчетов.

Запросы с принудительными планами

В отчете Queries With Forced Plans показаны запросы, для которых принудительно задан план выполнения.

Статистика ожидания запроса

Отчет Query Wait Statistics позволяет обнаруживать запросы с большими ожиданиями. Данные сгруппированы по нескольким категориям (таким, как ЦП, диск и блокировка) в зависимости от типа ожидания. В документации Microsoft[1] можно найти подробную информацию о том, что означает каждый тип ожидания.

Отслеживаемые запросы

Отчет Tracked Queries позволяет отслеживать планы выполнения и статистики для отдельных запросов. Он содержит ту же информацию, что и отчеты Regressed Queries и Top Resource Consuming Queries, но на уровне отдельных запросов.

Все эти отчеты предоставляют большой объем данных для анализа. Однако время от времени приходится использовать T-SQL и напрямую работать с данными хранилища запросов. Давайте посмотрим, как этого добиться.

Работа с динамическими административными представлениями хранилища запросов

Динамические административные представления хранилища запросов (Query Store DMVs) сильно нормализованы, как показано на рис. 4.9. Статистика выполнения отслеживается для каждого плана выполнения и группируется по интервалам сбора, которые задаются параметром `INTERVAL_LENGTH_MINUTES`. В большинстве случаев годится заданный по умолчанию интервал в 60 минут.

Нетрудно догадаться, что чем меньше интервалы, тем больше данных будет собрано в хранилище запросов. То же самое относится к рабочей нагрузке системы: чрезмерное количество нерегламентированных запросов может раздуть объем хранилища. Не забывайте об этом, когда настраиваете хранилище запросов в своей системе.

---

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

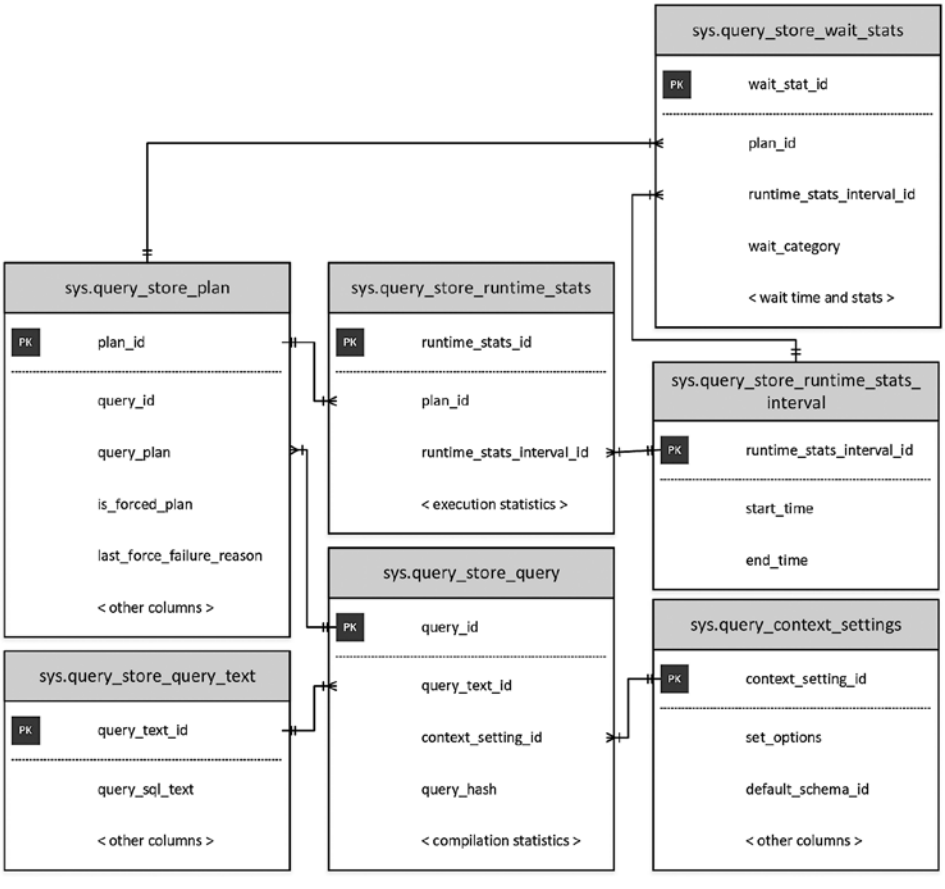


Рис. 4.9. Динамические административные представления хранилища запросов

Динамические административные представления логически относятся либо к хранилищу планов, либо к статистике времени выполнения. Представления хранилища планов таковы:

`sys.query_store_query`[1]

Информация о запросах и статистика их компиляции, а также время последнего запуска.

`sys.query_store_query_text`[2]

Сведения о тексте запроса.

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

[2] https://oreil.ly/Add1d

`sys.query_context_setting`[1]

Настройки контекста, связанные с запросом. Это представление содержит параметры `SET`, схему сеанса по умолчанию, язык и другие атрибуты. При разных наборах этих настроек SQL Server может генерировать и кэшировать разные планы выполнения для одного и того же запроса.

`sys.query_store_plan`[2]

Информация о планах выполнения запросов. Столбец `is_forced_plan` указывает, является ли план принудительным. Столбец `last_force_failure_reason` говорит, почему принудительный план не был применен к запросу.

Как видите, у каждого запроса может быть по несколько записей в представлениях `sys.query_store_query` и `sys.query_store_plan`. Это зависит от параметров контекста вашего сеанса, перекомпиляции и других факторов.

Три других представления относятся к статистике времени выполнения:

`sys.query_store_runtime_stats_interval`[3]

Информация об интервалах сбора статистики.

`sys.query_store_runtime_stats`[4]

Это представление ссылается на `sys.query_store_plan` и содержит информацию о статистике времени выполнения для того или иного плана в течение определенного интервала `sys.query_store_runtime_stats_interval`. Оно предоставляет сведения о количестве запусков, процессорном времени и длительности вызовов, статистике логических и физических операций ввода/вывода, использовании журнала транзакций, степени параллелизма, размере выделяемой памяти, а также некоторые другие полезные показатели.

`sys.query_store_wait_stats`[5]

Начиная с SQL Server 2017, с помощью этого представления можно получить информацию об ожиданиях запросов. Данные собираются для каждого плана и временного интервала и группируются по нескольким категориям ожидания, включая ЦП, память и блокировку.

Рассмотрим несколько сценариев работы с данными хранилища запросов.

---

[1] https://oreil.ly/gcu4z

[2] https://oreil.ly/EnY5C

[3] https://oreil.ly/Aa8Wy

[4] https://oreil.ly/Icoax

[5] https://oreil.ly/HmiJy

В листинге 4.8 показан код, который возвращает информацию о 50 запросах с самым интенсивным вводом/выводом. Поскольку хранилище запросов сохраняет статистику выполнения за множество отдельных временны́х интервалов, нужно агрегировать данные из нескольких строк `sys.query_store_runtime_stats`. В результате вы увидите данные для всех интервалов, которые закончились в течение последних 24 часов, сгруппированные по запросам и их планам выполнения.

Стоит отметить, что информация о дате и времени в хранилище запросов представлена типом `datetimeoffset`. Имейте это в виду, когда будете фильтровать данные.

Листинг 4.8. Получение информации о ресурсоемких запросах из хранилища запросов

```
SELECT TOP 50
  q.query_id, qt.query_sql_text, qp.plan_id, qp.query_plan
  ,SUM(rs.count_executions) AS [Execution Cnt]
  ,CONVERT(INT,SUM(rs.count_executions *
    (rs.avg_logical_io_reads + avg_logical_io_writes)) /
      SUM(rs.count_executions)) AS [Avg IO]
  ,CONVERT(INT,SUM(rs.count_executions *
    (rs.avg_logical_io_reads + avg_logical_io_writes))) AS [Total IO]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_cpu_time) /
    SUM(rs.count_executions)) AS [Avg CPU]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_cpu_time)) AS [Total CPU]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_duration) /
    SUM(rs.count_executions)) AS [Avg Duration]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_duration))
    AS [Total Duration]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_physical_io_reads) /
    SUM(rs.count_executions)) AS [Avg Physical Reads]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_physical_io_reads))
    AS [Total Physical Reads]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_query_max_used_memory) /
    SUM(rs.count_executions)) AS [Avg Memory Grant Pages]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_query_max_used_memory))
    AS [Total Memory Grant Pages]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_rowcount) /
    SUM(rs.count_executions)) AS [Avg Rows]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_rowcount)) AS [Total Rows]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_dop) /
    SUM(rs.count_executions)) AS [Avg DOP]
  ,CONVERT(INT,SUM(rs.count_executions * rs.avg_dop)) AS [Total DOP]
FROM
  sys.query_store_query q WITH (NOLOCK)
    JOIN sys.query_store_plan qp WITH (NOLOCK) ON
      q.query_id = qp.query_id
    JOIN sys.query_store_query_text qt WITH (NOLOCK) ON
      q.query_text_id = qt.query_text_id
    JOIN sys.query_store_runtime_stats rs WITH (NOLOCK) ON
      qp.plan_id = rs.plan_id
    JOIN sys.query_store_runtime_stats_interval rsi WITH (NOLOCK) ON
```

```
      rs.runtime_stats_interval_id = rsi.runtime_stats_interval_id
WHERE
  rsi.end_time >= DATEADD(DAY,-1,SYSDATETIMEOFFSET())
GROUP BY
  q.query_id, qt.query_sql_text, qp.plan_id, qp.query_plan
ORDER BY
  [Avg IO] DESC
OPTION (MAXDOP 1, RECOMPILE);
```

Очевидно, что данные можно сортировать не только по среднему количеству операций ввода/вывода, но и по другим критериям. Можно также добавить предикаты в предложения `WHERE` и/или `HAVING` запроса, чтобы ограничить результаты. Например, можно отфильтровать результаты по столбцам *DOP*, если требуется обнаружить запросы, которые используют параллелизм в среде OLTP, и тонко настроить параметр *Cost Threshold for Parallelism*.

Еще один пример — поиск запросов, которые раздувают кэш планов. Код в листинге 4.9 содержит информацию о запросах, которые создают несколько планов выполнения из-за разных настроек контекста. Чаще всего это происходит в двух случаях: либо в сеансах используются различные параметры `SET`, либо запросы ссылаются на объекты без имен схем.

Листинг 4.9. Запросы с разными настройками контекста

```
SELECT
    q.query_id, qt.query_sql_text
    ,COUNT(DISTINCT q.context_settings_id) AS [Context Setting Cnt]
    ,COUNT(DISTINCT qp.plan_id) AS [Plan Count]
FROM
    sys.query_store_query q WITH (NOLOCK)
        JOIN sys.query_store_query_text qt WITH (NOLOCK) ON
            q.query_text_id = qt.query_text_id
        JOIN sys.query_store_plan qp WITH (NOLOCK) ON
            q.query_id = qp.query_id
GROUP BY
    q.query_id, qt.query_sql_text
HAVING
    COUNT(DISTINCT q.context_settings_id) > 1
ORDER BY
    COUNT(DISTINCT q.context_settings_id)
OPTION (MAXDOP 1, RECOMPILE);
```

В листинге 4.10 показано, как найти похожие запросы по значению `query_hash` (SQL в выходных данных отображает один случайно выбранный запрос из каждой группы). Обычно эти запросы относятся к непараметризованной нерегламентированной рабочей нагрузке в системе. В коде эти запросы можно параметризовать. Если это не удается, попробуйте принудительную параметризацию, о которой я расскажу в главе 6.

Листинг 4.10. Обнаружение запросов с одинаковыми значениями `query_hash`

```
;WITH Queries(query_hash, [Query Count], [Exec Count], qtid)
AS
(
    SELECT TOP 100
        q.query_hash
        ,COUNT(DISTINCT q.query_id)
        ,SUM(rs.count_executions)
        ,MIN(q.query_text_id)
    FROM
        sys.query_store_query q WITH (NOLOCK)
            JOIN sys.query_store_plan qp WITH (NOLOCK) ON
                q.query_id = qp.query_id
            JOIN sys.query_store_runtime_stats rs WITH (NOLOCK) ON
                qp.plan_id = rs.plan_id
    GROUP BY
        q.query_hash
    HAVING
        COUNT(DISTINCT q.query_id) > 1
)
SELECT
    q.query_hash
    ,qt.query_sql_text AS [Sample SQL]
    ,q.[Query Count]
    ,q.[Exec Count]
FROM
    Queries q CROSS APPLY
    (
        SELECT TOP 1 qt.query_sql_text
        FROM sys.query_store_query_text qt WITH (NOLOCK)
        WHERE qt.query_text_id = q.qtid
    ) qt
ORDER BY
    [Query Count] DESC, [Exec Count] DESC
OPTION(MAXDOP 1, RECOMPILE);
```

В общем, возможности хранилища запросов безграничны. Используйте его, если ваша система способна выдержать сопутствующую нагрузку.

Наконец, команда `DBCC CLONEDATABASE`[1] позволяет создать клон базы данных, содержащий только схему, чтобы с его помощью исследовать проблемы с производительностью. По умолчанию клон включает в себя данные хранилища запросов, так что его можно восстановить и выполнить анализ на другом сервере, чтобы не перегружать промышленный экземпляр.

[1] https://oreil.ly/2jYGV

Сторонние инструменты

Как вы убедились, SQL Server предлагает богатый набор инструментов, чтобы обнаруживать неэффективные запросы. Тем не менее иногда бывают полезны и средства мониторинга от других поставщиков. Большинство из этих средств перечисляет самые ресурсоемкие запросы для анализа и оптимизации. Многие сторонние инструменты также способны задать отправную точку для анализа тенденций и обнаружения регрессивных запросов.

Я не буду описывать конкретные инструменты, но хочу предложить несколько советов о том, как их выбирать и использовать.

Чтобы эффективно применять тот или иной инструмент, в нем нужно разбираться. Изучите, как он работает, каковы его ограничения и какие данные он может упустить. Например, если инструмент получает данные, опрашивая представление `sys.dm_exec_requests` по расписанию, он может пропустить множество мелких, но часто выполняемых запросов, которые запускаются между опросами. Вместе с тем, если инструмент определяет неэффективные запросы по времени ожидания сеанса, то результаты будут сильно зависеть от рабочей нагрузки системы, объема кэшированных данных в буферном пуле и многих других факторов.

В зависимости от ваших конкретных потребностей эти ограничения могут быть приемлемыми. Помните принцип Парето: не обязательно оптимизировать абсолютно все неэффективные запросы в системе, чтобы добиться желаемой (или приемлемой) окупаемости. Всегда полезно взглянуть на проблему в целом и с разных сторон. Например, легко сверить список неэффективных запросов, который выводит инструмент, со статистикой выполнения на основе кэша планов и таким образом получить более полный список.

Разбираться в инструменте важно еще затем, чтобы оценивать накладные расходы, которые он может вызвать. Некоторые динамические административные представления очень ресурсоемки. Например, если инструмент вызывает функцию `sys.dm_exec_query_plan` при каждом опросе `sys.dm_exec_requests`, это может ощутимо увеличить нагрузку и без того высоконагруженной системы. Также нередко инструменты плодят трассировки и сеансы расширенных событий без вашего ведома.

Не стоит слепо доверять официальной документации и поставщикам, когда они заявляют, что инструмент безопасен. В разных системах он может проявить себя по-разному. Имеет смысл протестировать накладные расходы на рабочей нагрузке, сравнив показатели системы с инструментом и без него. Имейте в виду, что накладные расходы не обязательно постоянны и могут увеличиваться по мере изменения рабочей нагрузки.

Наконец, разберитесь, как выбранные инструменты влияют на безопасность. Многие инструменты позволяют создавать настраиваемые мониторы, которые выполняют запросы на сервере, открывая лазейки для вредоносной активности. Не предоставляйте лишних разрешений учетной записи инструмента и контролируйте, кто имеет доступ к управлению им.

В общем, выберите подход, который лучше всего поможет вам выявлять неэффективные запросы и лучше всего работает с вашей системой. Помните, что оптимизация запросов полезна в любой системе.

Резюме

Неэффективные запросы пагубно влияют на производительность SQL Server и могут перегрузить дисковую подсистему. Даже если в системе достаточно памяти, чтобы кэшировать данные в буферном пуле, эти запросы расходуют ресурсы ЦП, плодят блокировки и снижают качество обслуживания клиентов.

SQL Server отслеживает показатели выполнения каждого кэшированного плана и выводит их в представлении `sys.dm_exec_query_stats`. Также можно получать статистику выполнения для хранимых процедур, триггеров и пользовательских скалярных функций с помощью представлений `sys.dm_exec_procedure_stats`, `sys.dm_exec_trigger_stats` и `sys.dm_exec_function_stats` соответственно.

Статистика выполнения на основе кэша планов не отслеживает показатели времени выполнения для планов, а также запросы, планы которых не кэшировались. Учитывайте это, когда анализируете и настраиваете запросы.

Неэффективные запросы можно захватывать в режиме реального времени с помощью расширенных событий и трассировок SQL. Оба подхода создают существенные накладные расходы, особенно в высоконагруженных системах. Кроме того, они выводят сырые данные, которые придется обрабатывать и агрегировать для дальнейшего анализа.

В SQL Server 2016 и более поздних версиях можно использовать хранилище запросов. Это отличный инструмент, который не зависит от кэша планов и позволяет быстро выявлять регрессии плана. Хранилище запросов тоже добавляет накладные расходы, но они обычно приемлемы (хотя лучше оценить их в каждом конкретном случае).

Наконец, неэффективные запросы можно искать с помощью сторонних инструментов мониторинга. Важно разбираться в том, как работает тот или иной инструмент, и представлять себе его ограничения и накладные расходы.

В следующей главе я расскажу о распространенных методах, с помощью которых можно оптимизировать неэффективные запросы.

Чек-лист устранения неполадок

- ☐ Получить перечень неэффективных запросов из представления `sys.dm_exec_query_stats`. Отсортировать данные в соответствии с вашей стратегией устранения неполадок (ЦП, ввод/вывод и т. д.).
- ☐ Определить самые ресурсоемкие хранимые процедуры, используя представление `sys.dm_exec_procedure_stats`.
- ☐ По возможности включить хранилище запросов и анализировать данные, собранные с его помощью. (Это не всегда возможно, если вы уже используете внешние инструменты мониторинга.)
- ☐ Включить флаги трассировки `T7745` и `T7752`, чтобы повысить производительность запуска и завершения работы SQL Server, когда используется хранилище запросов.
- ☐ Проанализировать данные сторонних инструментов мониторинга и сверить их с данными SQL Server.
- ☐ Проанализировать накладные расходы, возникающие из-за неэффективных запросов. Изучить, как запросы потребляют ресурсы, и сверить эти показатели со статистикой ожидания и нагрузкой на сервер.
- ☐ При необходимости оптимизировать запросы.

ГЛАВА 5

Хранение данных и настройка запросов

Оптимизации и настройке запросов можно посвятить целую книгу. И таких книг уже много, так что советую почитать их всем, кто хочет улучшить свои навыки. Не пытаясь дублировать другие книги, в этой главе я рассмотрю некоторые из наиболее важных концепций, относящихся к настройке запросов.

Нельзя освоить оптимизацию запросов, если не разбираться во внутренней структуре индекса и принципах, которые SQL Server использует для доступа к данным. Поэтому глава начинается с общего обзора индексов на основе B-деревьев, а также операций поиска и просмотра.

Затем мы обсудим статистику и оценку количества элементов, а также узнаем, как читать и анализировать планы выполнения.

Наконец, я расскажу о распространенных проблемах, с которыми можно столкнуться в процессе настройки запросов, и объясню, как их решать и как индексировать данные.

Хранение данных и схемы доступа

Современные версии SQL Server поддерживают три технологии хранения и обработки данных. Самый старый и востребованный метод — *хранение на основе строк (row-based storage)*. При таком методе хранения все столбцы таблицы объединяются в строки данных, которые размещаются на страницах данных размером 8 Кбайт. Логически эти строки представлены либо индексами на основе B-деревьев, либо кучами (мы обсудим эти структуры чуть позже).

Начиная с SQL Server 2012, некоторые индексы или целые таблицы можно хранить в формате столбцов, используя *хранилище на основе столбцов (column-based storage)* и *индекс columnstore*. Данные в таких индексах сильно сжаты и хранятся для каждого столбца отдельно. Эта технология оптимизирована так, чтобы обес-

печивать высокую производительность в случае аналитических запросов только для чтения, когда просматриваются большие объемы данных. К сожалению, она плохо масштабируется в рабочей нагрузке OLTP.

Наконец, начиная с SQL Server 2014, доступен механизм *In-Memory OLTP*, который хранит данные в *таблицах, оптимизированных для памяти*. Данные в таких таблицах полностью размещаются в памяти и отлично подходят для высоких рабочих нагрузок OLTP.

Можно использовать все три технологии вместе — таблицы на основе строк, таблицы на основе столбцов и таблицы, оптимизированные для памяти, — распределив данные между ними. Этот подход чрезвычайно полезен, когда в одной и той же системе нужно поддерживать и ресурсоемкие OLTP, и аналитические рабочие нагрузки. Эта архитектура подробно описана в моей книге *Pro SQL Server Internals* (Apress, 2016).

По умолчанию используется хранение на основе строк, и это, безусловно, самый распространенный формат в SQL Server. Операторы `CREATE TABLE` и `CREATE INDEX` сохраняют данные в строках, если явно не указано иное. Хранение на основе строк хорошо справляется с умеренными рабочими нагрузками OLTP и аналитическими задачами. Оно добавляет меньше административных издержек, чем индексы columnstore и In-Memory OLTP.

В этой главе мы поговорим о хранении на основе строк и запросах, которые работают с индексами на основе B-деревьев. Начнем с того, как SQL Server хранит данные на основе строк.

Таблицы на основе строк

Внутренняя структура таблицы на основе строк состоит из нескольких элементов и внутренних объектов, как показано на рис. 5.1.

Данные в таблицах либо хранятся полностью несортированными (такие таблицы называются *кучами*), либо сортируются по значению ключа кластеризованного индекса, если он определен.

Не углубляясь в детали, скажу лишь, что обычно лучше не хранить данные в кучах, а определить для таблиц кластеризованные индексы. Только в некоторых особых случаях таблицы-кучи оказываются эффективнее кластеризованных индексов.

Помимо одного кластеризованного индекса, у каждой таблицы может быть набор *некластеризованных индексов*: это отдельные структуры, в которых хранятся копии данных из таблицы, отсортированные по ключевым столбцам индекса. Например, если столбец включен в два некластеризованных индекса, то SQL Server

сохранит эти данные трижды: один раз в кластеризованном индексе или куче и по одному разу в каждом из двух некластеризованных индексов.

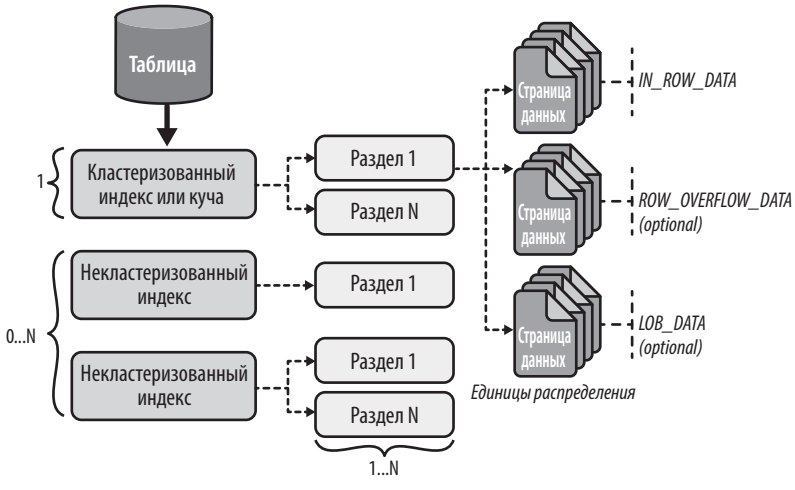


Рис. 5.1. Внутренняя структура таблицы на основе строк

Хотя SQL Server позволяет создавать много некластеризованных индексов, лучше так не делать, особенно в системах OLTP с изменчивыми данными. Это чревато не только накладными расходами на хранение, но и тем, что при модификации данных SQL Server будет вставлять, обновлять или удалять строки в каждом некластеризованном индексе, поддерживая несколько копий данных.

Большое количество некластеризованных индексов — плохая идея, но это не значит, что вам стоит немедленно искоренять лишние индексы. У таблицы должно быть столько индексов, чтобы нагрузка системы была оптимальна.

В главе 14 я расскажу, как анализировать использование индексов.

Всякий индекс (как и куча) состоит из одного или нескольких разделов. Каждый раздел — это внутренняя структура данных (индекс или куча), независимая от других разделов таблицы. Каждый индекс в таблице можно разбивать на разделы по-своему, хотя обычно полезно разбивать все индексы одинаково, чтобы они были выровнены относительно друг друга.

Как я упоминал ранее, фактические данные хранятся в строках данных на страницах данных размером 8 Кбайт, из которых пользователям доступно 8060 байт. Данные из всех столбцов хранятся вместе, кроме случаев, когда данные столбца не помещаются на странице.

Страницы данных делятся на три группы, называемые *единицами распределения*.

На страницах единиц распределения `IN_ROW_DATA` хранятся строковые объекты основных данных: внутренние атрибуты и данные из столбцов фиксированной длины (таких, как `int`, `datetime`, `float` и т. д.). Внутренняя часть строки должна умещаться на одной странице данных, поэтому ее размер не может превышать 8060 байт. Данные из столбцов переменной длины, таких как `(n)var char(max)`, `varbinary(max)`, `xml` и других, тоже могут храниться в основных строковых объектах, если удовлетворяют этому ограничению.

Когда данные переменной длины не помещаются в строке, SQL Server сохраняет их вне строки на других страницах данных и ссылается на них с помощью указателей в строке. Данные переменной длины, размер которых превышает 8000 байт, размещаются на страницах данных, относящихся к единицам распределения `LOB_DATA` (`LOB` означает «large objects» — «большие объекты»). В остальных случаях данные размещаются на страницах единиц распределения `ROW_OVERFLOW_DATA`.

Данные типа `(n)text` и `image` по умолчанию хранятся в единицах распределения `LOB_DATA`. Эту настройку можно регулировать с помощью хранимой процедуры `sp_tableoption`[1].

Здесь стоит повторить известный совет: *не извлекайте* в инструкциях `SELECT` ненужные столбцы, особенно с помощью `SELECT *`. Для этого могут понадобиться дополнительные операции ввода/вывода, чтобы получить данные со страниц вне строки, а использование покрывающих индексов будет отложено, как вы увидите далее в этой главе.

Наконец, SQL Server логически группирует блоки по 8 страниц в единицы распределения по 64 Кбайт, называемые *экстентами*. Существуют два типа экстентов. В *смешанных (mixed) экстентах* хранятся данные, относящиеся к разным объектам. *Однородные (uniform) экстенты* хранят данные для одного и того же объекта. По умолчанию при создании нового объекта SQL Server сохраняет первые 8 страниц объекта в смешанных экстентах. После этого все дополнительное пространство для этого объекта выделяется однородными экстентами.

Выделение смешанных экстентов можно отключить с помощью флага трассировки `T1118` на уровне сервера. В SQL Server 2016 и более поздних версиях можно управлять выделением смешанных экстентов на уровне базы данных с помощью параметра `MIXED_PAGE_ALLOCATION`. Если отключить смешанные экстенты, то при создании каждой новой таблицы понадобится меньше изменений в системных таблицах. Обычно это не дает заметных преимуществ в пользовательских базах данных, но может значительно улучшить пропускную способность базы данных

[1] https://oreil.ly/7WTZi

tempdb в высоконагруженных системах OLTP. Отключать выделение смешанных экстентов с помощью флага трассировки T1118 имеет смысл в старых версиях SQL Server (до 2016). Начиная с SQL Server 2016, смешанные экстенты в tempdb уже не используются, поэтому этот флаг неактуален.

Теперь посмотрим на структуру индексов.

Индексы на основе B-деревьев

Кластеризованные и некластеризованные индексы хранятся в очень похожих внутренних форматах *B-деревьев*. Для примера создадим таблицу Customers, как показано в листинге 5.1. В таблице заданы индексы: кластеризованный индекс для столбца CustomerId и некластеризованный — для столбца Name.

Листинг 5.1. Таблица Customers

```
CREATE TABLE dbo.Customers
(
    CustomerId INT NOT NULL,
    Name NVARCHAR(64) NOT NULL,
    Phone VARCHAR(32) NULL,
    /* Прочие столбцы */
);

CREATE UNIQUE CLUSTERED INDEX IDX_Customers_CustomerId
ON dbo.Customers(CustomerId);

CREATE NONCLUSTERED INDEX IDX_Customers_Name
ON dbo.Customers(Name);
```

ОГРАНИЧЕНИЯ ИЛИ ИНДЕКСЫ?

Как вы могли заметить, я определил для таблицы кластеризованный индекс, вместо того чтобы создать ограничение типа «первичный ключ». Это сделано намеренно. Я всегда рассматриваю ограничения как часть логической структуры базы данных, которая определяет сущности и их ключевые атрибуты. Индексы же относятся к физической структуре базы данных.

По умолчанию SQL Server создает уникальные кластеризованные индексы для первичных ключей. Однако можно, а часто даже нужно помечать первичные ключи как некластеризованные, чтобы они стали уникальными некластеризованными индексами.

За исключением нескольких ситуаций, когда SQL Server требует задавать первичные ключи, выбор между ограничениями и индексами — дело вкуса. Ограничения Primary (*первичный*) и Unique (*уникальный*) на внутреннем уровне реализованы как индексы и ведут себя соответствующим образом. Во время настройки производительности мы будем работать с индексами, поэтому в этой книге ограничениям не уделяется специального внимания. Но это не означает, будто вам не надо их использовать.

Логическая структура кластеризованного индекса показана на рис 5.2.

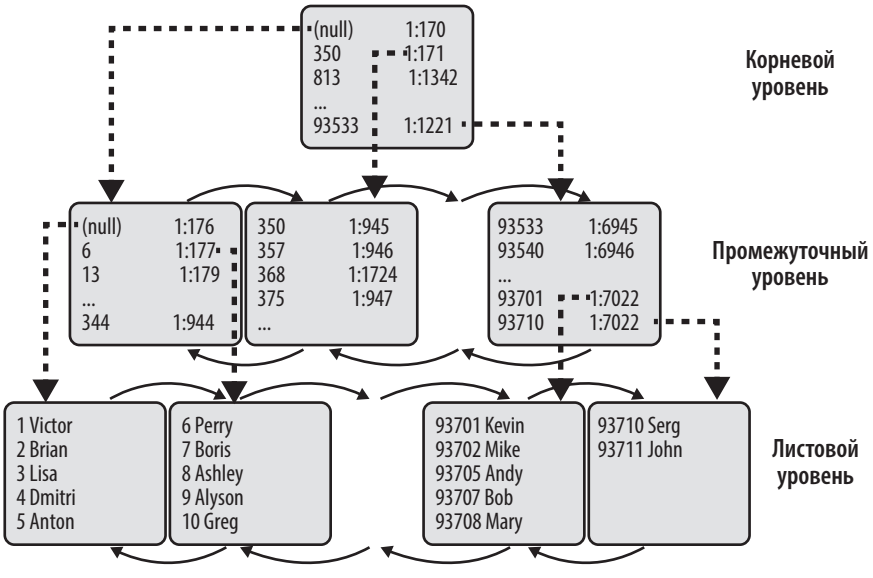


Рис. 5.2. Индекс на основе B-дерева

Нижний уровень индекса называется *листовым*. На нем хранятся данные, отсортированные по ключу индекса. Если индекс кластеризованный, то на листовом уровне хранятся все данные из таблицы, отсортированные по кластеризованному ключу. Если быть точнее, листовой уровень содержит только данные `IN_ROW`, которые могут ссылаться на данные столбца вне строки, размещенные на других страницах.

Если все данные в индексе умещаются на одной странице данных, то индекс будет состоять из этой единственной листовой страницы. В противном случае SQL Server начнет создавать *промежуточные уровни* индекса. Каждая строка на странице промежуточного уровня ссылается на страницу уровнем ниже и содержит минимальное значение ключа на той странице, а также ее физический адрес (`FileId:PageId`) в базе данных. Единственное исключение — самая первая строка, в которой хранится `NULL` вместо минимального значения ключа.

SQL Server создает промежуточные уровни, пока не достигнет уровня с единственной страницей. Этот уровень называется *корневым* и служит точкой входа в индекс.

Страницы на каждом уровне индекса объединены в двусвязный список. Каждая страница ссылается на предыдущую и следующую страницы в индексе. Это позволяет SQL Server просматривать индексы вперед и назад. (Однако

имейте в виду, что обратный просмотр может быть менее эффективным, потому что в этой операции SQL Server не использует параллелизм.)

SQL Server может получить доступ к данным в индексе посредством *просмотра индекса* или *поиска по индексу*. Просмотр индекса осуществляется двумя способами.

Первый способ — *просмотр порядка распределения (allocation order scan)*. С помощью системных страниц, называемых *картами распределения индексов* (IAM, Index Allocation Map), SQL Server отслеживает экстенты, относящиеся к каждому индексу в базе данных. Он в случайном порядке считывает страницы данных из индекса в соответствии с IAM. В SQL Server этот метод используется только в особых случаях, потому что может вызвать проблемы с согласованностью данных.

Второй, более распространенный метод называется *упорядоченным просмотром (ordered scan)*. Предположим, вы запускаете запрос `SELECT Name FROM dbo.Customers`. Все строки данных находятся на листовом уровне индекса, так что SQL Server может просмотреть этот уровень и вернуть строки клиенту.

SQL Server начинает с корневой страницы индекса и считывает оттуда первую строку. Эта строка ссылается на промежуточную страницу с минимальным значением ключа из таблицы. SQL Server считывает эту страницу и повторяет процесс, пока не дойдет до первой страницы на листовом уровне. Затем SQL Server считывает строки одну за другой, перемещаясь по связанному списку страниц, пока не прочитает все строки (рис. 5.3).

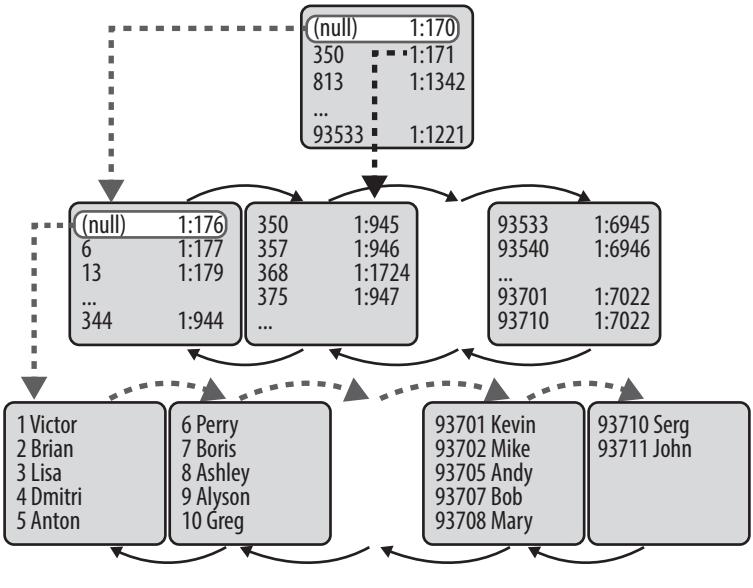


Рис. 5.3. Просмотр индекса

Очевидно, в реальной жизни все может оказаться сложнее. Например, в некоторых случаях запрос может одновременно просматривать разные части индекса в соответствии с параллельными планами выполнения. А когда несколько запросов выполняются одновременно, SQL Server может объединять соответствующие просмотры индекса в один просмотр физического индекса. В любом случае, если в плане выполнения есть оператор *Index Scan*, можно предположить, что он извлечет все данные из индекса.

При этом есть одно исключение — когда в плане просмотр индекса идет сразу после оператора `TOP`. В этом случае оператор просмотра остановится после того, как вернет количество строк, указанное в `TOP`, и не будет перебирать всю таблицу. Обычно так происходит, если в запросе нет предложения `ORDER BY` или если `ORDER BY` соответствует ключу индекса.

На рис. 5.4 показана часть плана выполнения запроса `SELECT TOP 3 Name FROM dbo.Customers ORDER BY CustomerId`. Свойства *Actual Rows Read* и *Actual Rows* оператора *Index Scan* показывают, что просмотр останавливается после того, как прочитаны три строки.

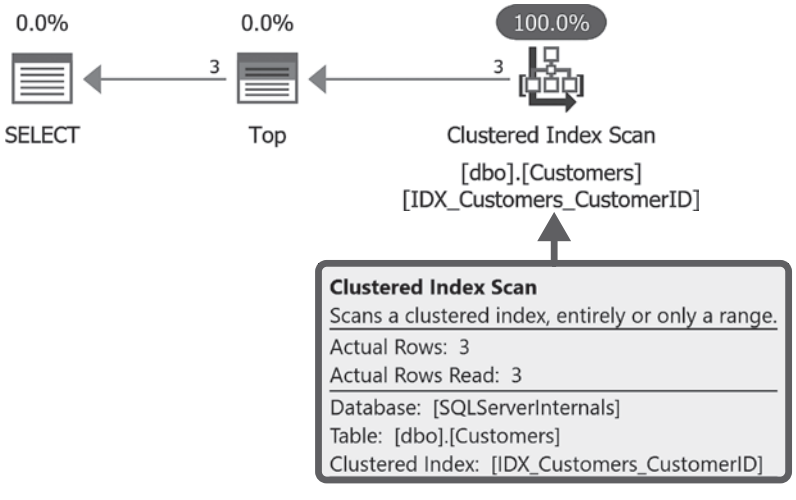


Рис. 5.4. Операторы *Top* и *Index Scan*

Как вы наверняка догадались, чтение всех данных из большого индекса — затратная операция. К счастью, SQL Server может обратиться к подмножеству данных, используя операцию *поиска по индексу*. Допустим, вы запускаете запрос `SELECT Name FROM dbo.Customers WHERE CustomerId BETWEEN 4 AND 7`. На рис. 5.5 показано, как SQL Server может его обработать.

Чтобы прочитать диапазон строк из таблицы, SQL Server должен найти строку с минимальным значением ключа из диапазона (в данном случае 4). SQL Server

начинает с корневой страницы, где вторая строка ссылается на страницу с минимальным значением ключа 350. Это больше, чем искомое значение ключа, поэтому SQL Server считывает страницу данных промежуточного уровня (1:170), на которую ссылается первая строка корневой страницы.

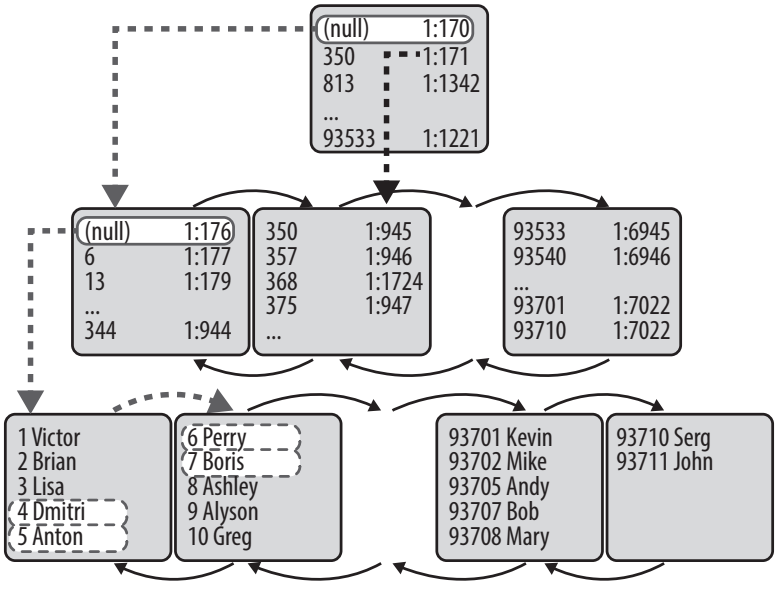


Рис. 5.5. Оператор Index Seek (Поиск по индексу)

Аналогично промежуточная страница приводит к первой листовой странице (1:176). SQL Server считывает эту страницу, затем строки с `CustomerId`, равными 4 и 5, и, наконец, считывает две оставшиеся строки со второй страницы.

Формально существует два вида операций поиска по индексу:

Точечный поиск

При точечном поиске (также называемом одноэлементным поиском) SQL Server ищет и возвращает одну строку. Например, предикат `WHERE CustomerId = 2` — это операция точечного поиска.

Просмотр диапазона

При просмотре диапазона SQL Server ищет наименьшее или наибольшее значение ключа и просматривает набор строк (вперед или назад), пока диапазон не кончится. Предикат `WHERE CustomerId BETWEEN 4 AND 7` приводит к просмотру диапазона. В планах выполнения оба случая фигурируют как операторы *Index Seek*.

Нетрудно догадаться, что поиск по индексу более эффективен, чем просмотр индекса, потому что при поиске по индексу SQL Server обычно обрабатывает лишь подмножество строк и страниц данных, а не обходит весь индекс. Однако оператор *Index Seek* в плане выполнения может сбивать с толку: это происходит, когда он обозначает неэффективный просмотр диапазона, при котором считывается большое количество строк или даже весь индекс. О такой ситуации мы поговорим позже в этой главе.

В реляционных базах данных существует концепция *предикатов с поддержкой поиска и аргументов* (SARG, Search Argument-able). Такие предикаты позволяют изолировать подмножество ключа индекса для последующей обработки. Благодаря SARG-предикатам SQL Server может отобрать одно значение или диапазон значений ключа индекса, которые будут считаны при вычислении предиката, и использовать поиск по индексу, если индекс существует.

Очевидно, при составлении запросов лучше использовать предикаты с поддержкой поиска и аргументов, а также по возможности задействовать поиск по индексу. Это делается с помощью операторов, в том числе `=`, `>`, `>=`, `<`, `<=`, `IN`, `BETWEEN`, а также `LIKE` (с начала строки). К операторам, не поддерживающим SARG, относятся `NOT`, `<>`, `LIKE` (не с начала строки) и `NOT IN`.

Предикаты также не поддерживают SARG, когда системные или пользовательские (невстроенные) функции применяются к столбцам таблицы. Чтобы вычислить предикат, SQL-сервер должен вызвать функцию для каждой обрабатываемой строки. Это несовместимо с поиском по индексу.

То же самое относится к преобразованиям типов данных, когда SQL Server использует внутреннюю функцию `CONVERT_IMPLICIT`. Один из распространенных примеров — это когда юникодный параметр `nvarchar` в предикате используется со столбцом `varchar`, где применяется алгоритм сопоставления строк SQL. Другой пример — когда в предикате соединения участвуют столбцы разных типов (хотя некоторые типы данных и некоторые неявные преобразования не вызывают проблем). Оба случая могут привести к просмотру индекса, даже если оператор предиката выглядит как SARG-совместимый.

Составные индексы

Индексы с несколькими ключевыми столбцами называются *составными*. Данные в составных индексах сортируются по столбцам слева направо. На рис. 5.6 показана структура составного индекса, определенного для столбцов `LastName` и `FirstName` в таблице. Данные сортируются сначала по `LastName` (крайний левый столбец), а затем по `FirstName` для каждого значения `LastName`.

Поддержка поиска и аргументов в составном индексе зависит от того, есть ли эта поддержка у предикатов в крайних левых столбцах индекса, что позволяет SQL Server определять и изолировать диапазон ключей индекса для обработки.

В табл. 5.1 показаны примеры предикатов, поддерживающих и не поддерживающих SARG, с использованием индекса на рис. 5.6.

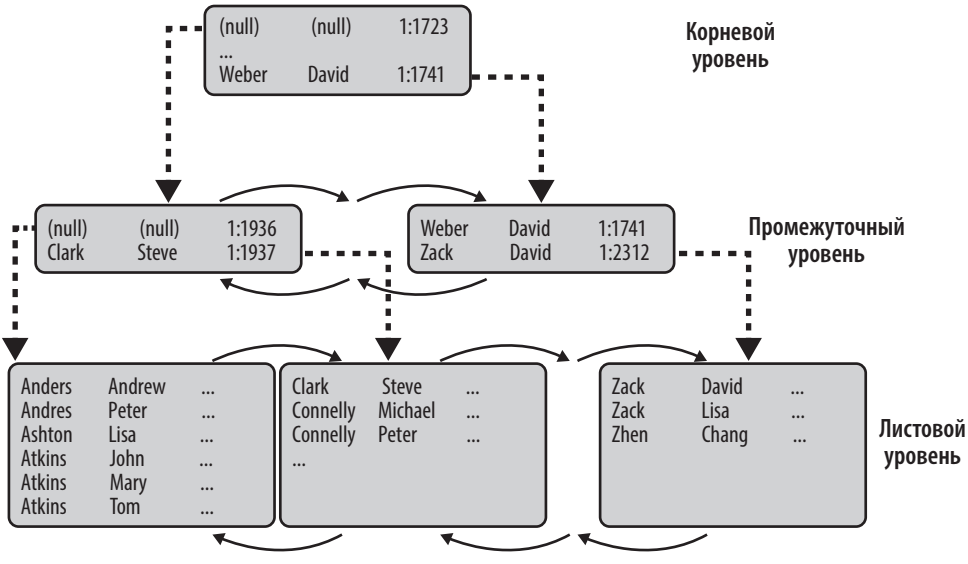


Рис. 5.6. Составные индексы

Таблица 5.1. Составной индекс и поддержка SARG

Предикаты, поддерживающие SARG	Предикаты, не поддерживающие SARG
`LastName = 'Clark' AND FirstName = 'Steve'`	`LastName <> 'Clark' AND FirstName = 'Steve'`
`LastName = 'Clark' AND FirstName <> 'Steve'`	`LastName LIKE '%ar%' AND FirstName = 'Steve'`
`LastName = 'Clark'`	`FirstName = 'Steve'`
`LastName LIKE 'Cl%'`	

Некластеризованные индексы

В отличие от кластеризованного индекса, который указывает, как строки данных сортируются в таблице, некластеризованный индекс задает отдельный порядок сортировки для столбца или набора столбцов, сохраняя их как отдельные структуры данных.

Для примера представьте себе обычную книгу. Номера страниц — это *кластеризованный* индекс книги. А в *предметном указателе* в конце книги перечислены

термины в алфавитном порядке, и каждый термин ссылается на номера страниц, где он упоминается. Таким образом, это *некластеризованный* индекс терминов.

Когда нужно найти термин в книге, вы ищете его в указателе. Это быстрая и эффективная операция, потому что термины отсортированы по алфавиту. Затем вы можете быстро найти нужные страницы по их номерам, указанным для каждого термина. Без предметного указателя оставалось бы лишь читать страницу за страницей, пока не найдутся все упоминания соответствующего термина.

Как я уже отмечал, кластеризованные и некластеризованные индексы устроены похожим образом в виде B-дерева. На рис. 5.7 показана структура некластеризованного индекса (вверху) для столбца `Name`, который мы создали в листинге 5.1. Внизу для сравнения приведен кластеризованный индекс.

Листовой уровень некластеризованного индекса сортируется по ключу индекса (`Name`). Каждая строка на листовом уровне содержит ключевое значение и *идентификатор строки*. Для таблиц с кластеризованным индексом этот идентификатор представляет собой ключ строки.

Очень важно запомнить: если в таблице определен кластеризованный индекс, то некластеризованные индексы *не* хранят информацию о физическом местоположении строки. Вместо этого в них хранится *значение ключа кластеризованного индекса*. Это также означает, что в некластеризованных индексах содержатся данные из ключевых столбцов кластеризованного индекса, *даже если вы явно не добавляете* эти столбцы при объявлении индекса.

Как и в случае с кластеризованным индексом, промежуточные и корневой уровни некластеризованного индекса содержат по одной строке для каждой страницы нижележащего уровня, на который они ссылаются. Эта строка состоит из физического адреса и минимального значения ключа со страницы. В неуникальных индексах также хранится идентификатор такой строки.

Давайте посмотрим, как в SQL Server применяются некластеризованные индексы. Выполним следующий запрос: `SELECT Name, Phone FROM dbo.Customers WHERE Name = 'Boris'`. Соответствующий процесс показан на рис. 5.8.

Как и в случае кластеризованного индекса, SQL Server начинает с корневой страницы некластеризованного индекса. Значение ключа *Boris* меньше, чем *Dan*, поэтому SQL Server переходит к промежуточной странице, на которую ссылается первая строка страницы корневого уровня.

Вторая строка промежуточной страницы указывает, что минимальное значение ключа на странице — *Boris*, хотя индекс не был объявлен как уникальный, и SQL Server не знает, есть ли на первой странице другие строки с ключом *Boris*. В результате сервер переходит на первую листовую страницу индекса и находит строку с ключом *Boris* и идентификатором 7.

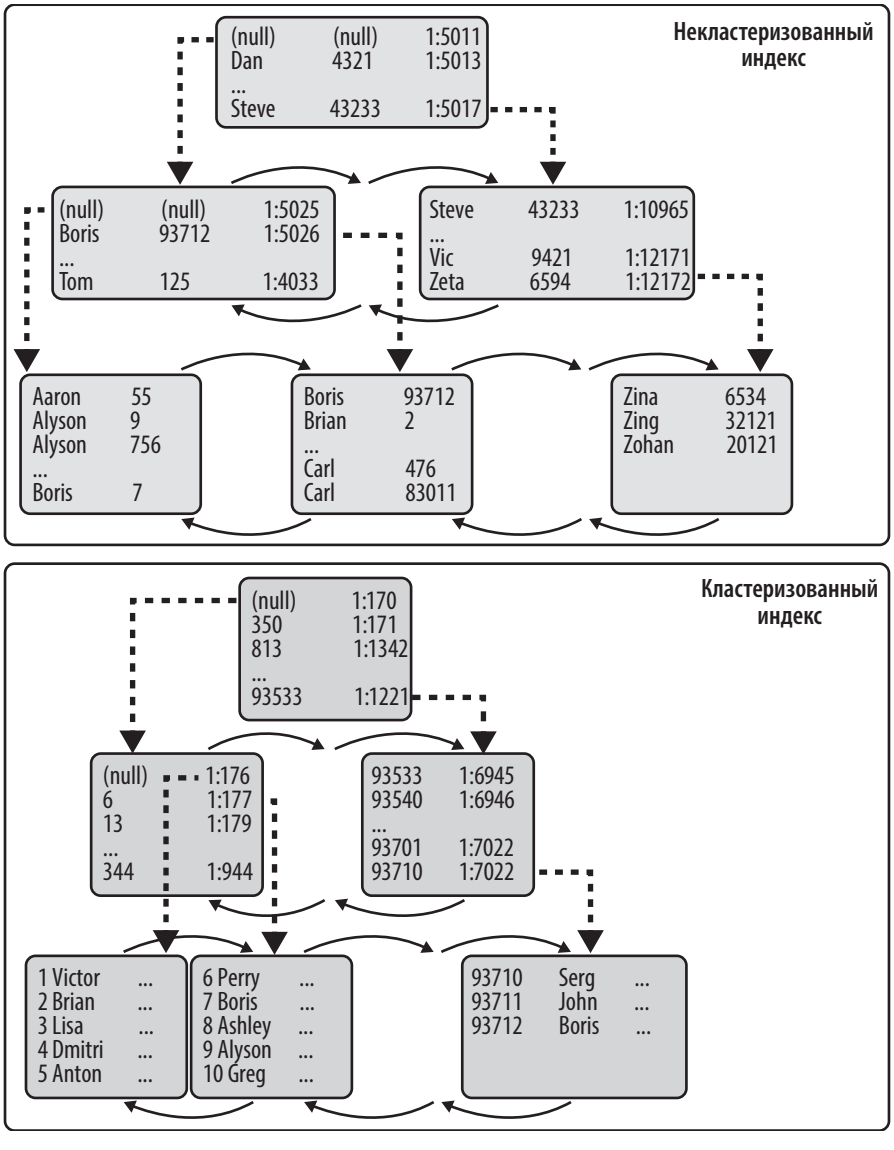


Рис. 5.7. Кластеризованный и некластеризованный индексы таблицы `Customers`

В нашем случае некластеризованный индекс хранит только значения `CustomerId` и `Name`, и SQL Server обходит дерево кластеризованного индекса, чтобы извлечь оттуда данные столбца `Phone`. Эта операция называется *поиском по ключу* (*key lookup*) или *поиском по идентификатору строки (RID lookup)* в кучах.

На следующем шаге, показанном на рис. 5.9, SQL Server возвращается к некластеризованному индексу и считывает вторую страницу с листового уровня. Он находит еще одну строку с ключом *Boris* и идентификатором 93712 и снова выполняет поиск по ключу.

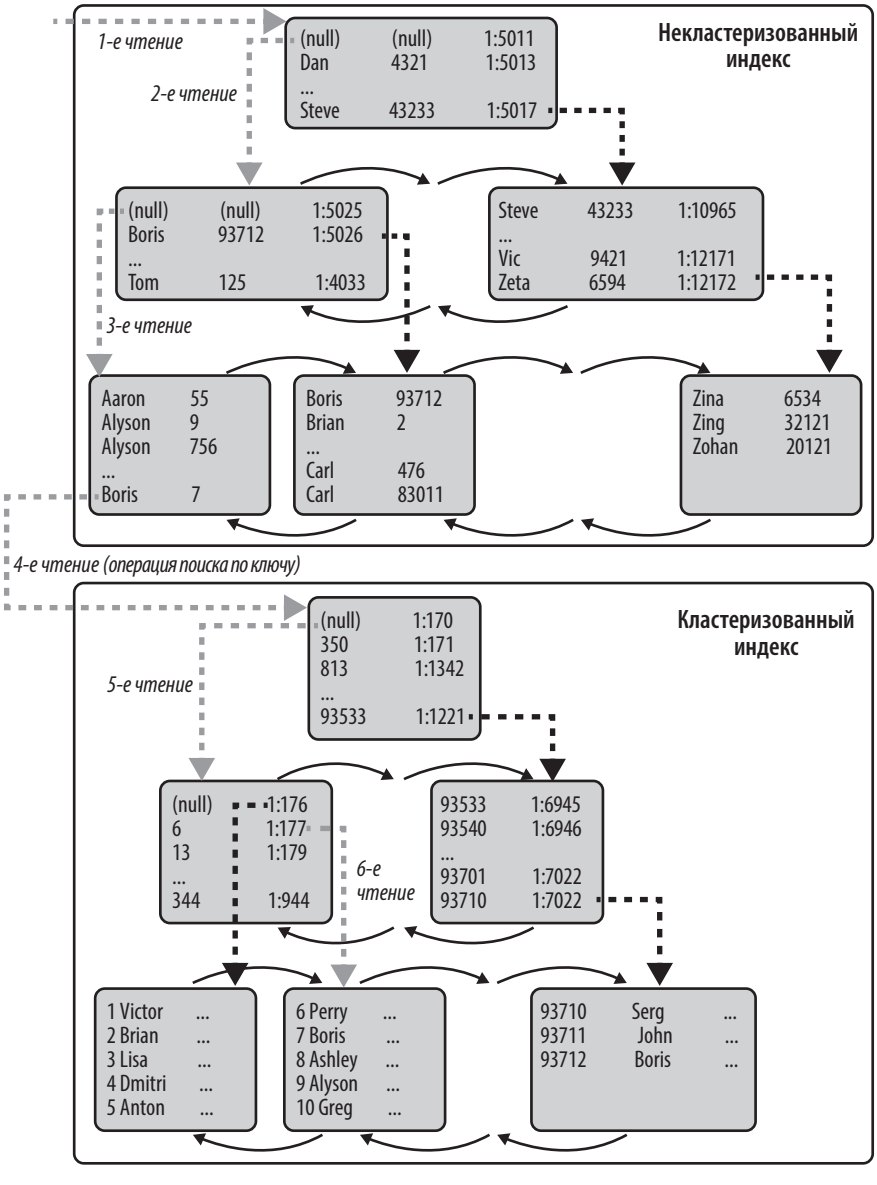


Рис. 5.8. Использование некластеризованного индекса, часть 1

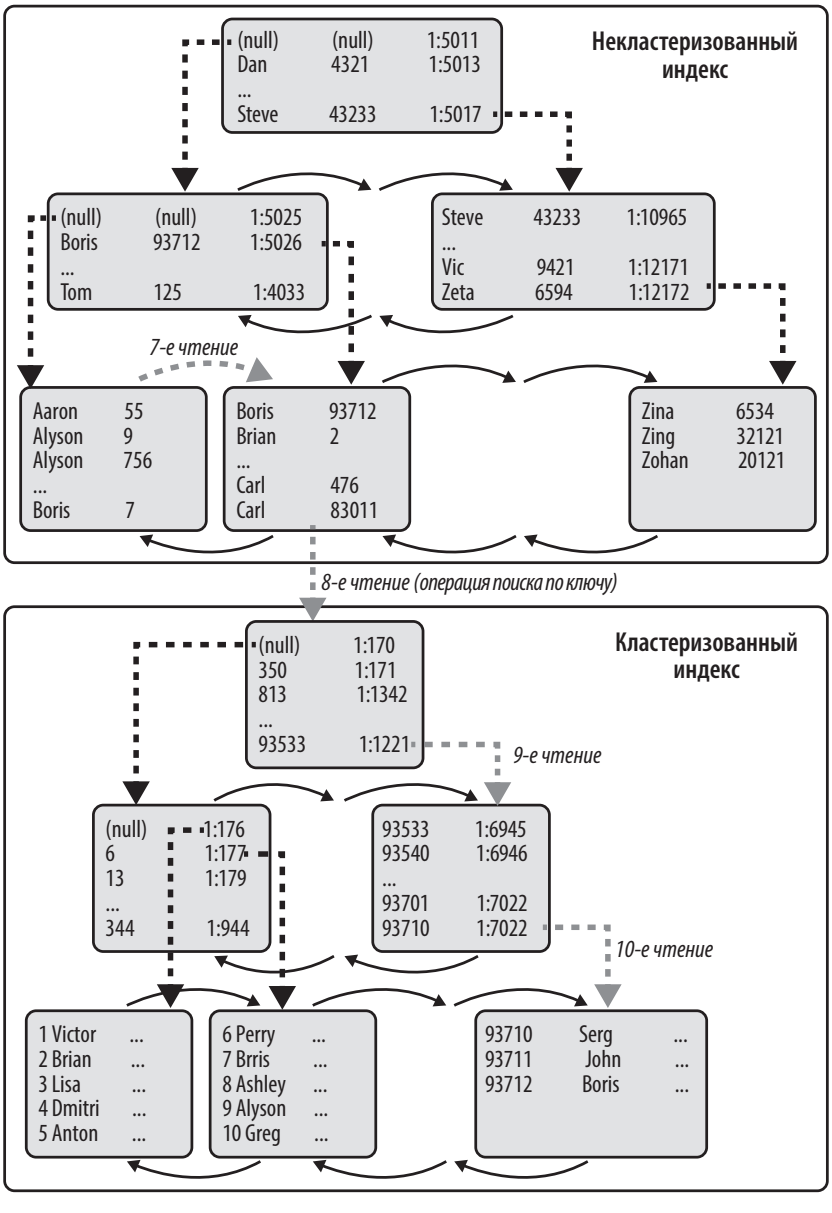


Рис. 5.9. Использование некластеризованного индекса, часть 2

Как видно на рис. 5.9, SQL Server выполняет 10 операций чтения, хотя запрос возвратил всего две строки. Количество операций ввода/вывода можно рассчитать по следующей формуле:

(количество уровней в некластеризованном индексе) + (количество страниц, считанных с листового уровня некластеризованного индекса) + (количество найденных строк) * (количество уровней в кластеризованном индексе).

Чем больше найдено строк (то есть чем больше операций поиска по ключу), тем больше требуется операций ввода/вывода, из-за чего некластеризованный индекс становится неэффективным.

Поэтому SQL Server очень консервативен при выборе некластеризованных индексов, когда ожидается, что потребуется много операций поиска по ключу. Вместо этого SQL Server может выполнить просмотр кластеризованного или другого некластеризованного индекса. Пороговое значение, при котором SQL Server перестает использовать некластеризованный индекс с поиском по ключу, может варьироваться, но в любом случае оно очень низкое — часто доля процента от общего количества строк в таблице.

В сопутствующие материалы книги входит сценарий, который демонстрирует, как проявляется описанная неэффективность и как SQL Server оптимизирует запросы.

То же самое относится к операциям поиска по идентификатору строки (RID). В кучах некластеризованные индексы в идентификаторе строки хранят ее физический адрес. Технически SQL Server может получить доступ к строке в куче за одну операцию чтения, но это все равно затратно. Более того: если при обновлении новая версия строки не помещается на старой странице данных, то SQL Server переместит ее в другое место, ссылаясь на нее через еще одну структуру, которая называется *указателем переадресации* и содержит адрес новой версии строки. Некластеризованные индексы будут по-прежнему ссылаться на указатели переадресации в идентификаторе строки, и поиск по нему может потребовать нескольких операций чтения для доступа к строке.

Фрагментация индекса

Хотя фрагментация индекса напрямую не относится к теме настройки запросов, ее нельзя не упомянуть, раз уж мы говорим об индексах. В конце концов, при настройке производительности SQL Server важно улучшать стратегию обслуживания индексов и уменьшать их фрагментацию.

SQL Server всегда поддерживает порядок данных в индексе, вставляя новые строки на страницы данных, которым они принадлежат. Если на странице данных недостаточно свободного места, SQL Server выделяет новую страницу и помещает туда строку, корректируя указатели в двусвязном списке страниц,

чтобы сохранить логический порядок сортировки в индексе. Эта процедура называется *разбиением страницы* и приводит к фрагментации индекса, как вы увидите в этом разделе.

Описанная ситуация проиллюстрирована на рис. 5.10. Когда на исходной странице не хватает места для новой строки, SQL Server разбивает страницу: он перемещает примерно половину данных с исходной страницы на новую, а после этого корректирует указатели на страницы.

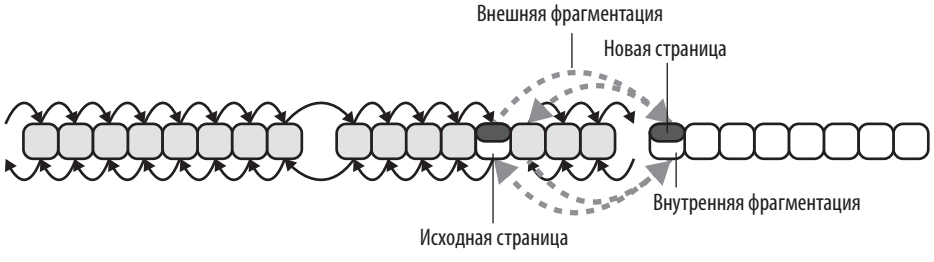


Рис. 5.10. Разбиение страницы

Разбиение страницы также может происходить при модификации данных. Когда обновление невозможно выполнить на месте — например, если размер строки данных увеличивается, — SQL Server разбивает страницу и перемещает обновленные и последующие строки с этой страницы на другую. Порядок сортировки индекса поддерживается с помощью указателей на страницы.

Существуют два вида фрагментации индекса.

Внешняя фрагментация

Внешняя фрагментация означает, что логический порядок страниц не соответствует их физическому порядку в файлах данных и/или что страницы, логически следующие друг за другом, не расположены в одном и том же или смежных экстентах. Внешняя фрагментация вынуждает SQL Server скакать туда-сюда при чтении данных с диска, отчего упреждающее чтение становится менее эффективным и требуется больше физических операций чтения. Быстродействие особенно страдает при использовании магнитных дисков, где произвольный ввод/вывод менее эффективен, чем последовательный.

Внутренняя фрагментация

Внутренняя фрагментация подразумевает, что на страницах данных в индексе есть свободное место. В результате индекс распространяется на большее количество страниц данных, а от этого, в свою очередь, растет количество логических операций чтения при выполнении запроса. Кроме того, SQL

Server использует больше памяти в буферном пуле, чтобы кэшировать страницы индекса.

Иногда небольшая степень внутренней фрагментации полезна. Она уменьшает разбиение страниц во время операций вставки и обновления, если данные модифицируются на разных страницах индекса. Но высокая степень внутренней фрагментации приводит к тому, что пространство, выделенное для индекса, используется нерационально и производительность системы снижается.

Проанализировать фрагментацию индексов можно с помощью функции `sys.dm_db_index_physical_stats`[1]. (Ее надо вызывать в режиме `DETAILED`, который требует, чтобы SQL Server просматривал всю таблицу.) Вот три самых важных столбца результата этой функции:

`avg_page_space_used_in_percent`

Показывает, какую долю страницы (в процентах) в среднем занимают данные. Это значение характеризует внутреннюю фрагментацию индекса.

`avg_fragmentation_in_percent`

Характеризует внешнюю фрагментацию индекса. Для таблиц с кластеризованными индексами это процент неупорядоченных страниц, когда следующая физическая страница, выделенная в индексе, отличается от той, на которую с текущей страницы ссылается указатель следующей страницы. Для кучи это процент неупорядоченных экстентов, для которых логический порядок следующего экстента не совпадает с физическим.

`fragment_count`

Количество непрерывных фрагментов данных в индексе. Каждый фрагмент представляет собой группу примыкающих друг к другу экстентов. Когда данные примыкают друг к другу, повышается вероятность, что при доступе к ним SQL Server будет использовать последовательный ввод/вывод и упреждающее чтение.

Недостатки фрагментированного индекса можно компенсировать современным оборудованием, когда на сервере достаточно памяти для кэширования данных в буферном пуле и есть быстрая подсистема ввода/вывода на основе флеш-памяти для чтения данных. Хотя уменьшать фрагментацию в системе всегда полезно, стоит оценить ее влияние, прежде чем разрабатывать стратегию обслуживания индексов.

Если у вашей системы есть периоды низкой активности (например, по ночам или в выходные дни), используйте это время для обслуживания индексов. Од-

[1] https://oreil.ly/YC90O

нако если система круглосуточно обрабатывает тысячи транзакций в секунду, проведите анализ и оцените преимущества и недостатки различных стратегий обслуживания индексов. Не забывайте, что обслуживание индекса — ресурсоемкая операция, которая увеличивает нагрузку на действующую систему.

Есть два метода обслуживания индекса, которые уменьшают фрагментацию: реорганизация индекса и его перестроение.

Реорганизация индекса

Реорганизация, или *дефрагментация*, индекса переупорядочивает страницы данных на листовом уровне согласно их логическому порядку. Также эта процедура пытается сжимать страницы, уменьшая их внутреннюю фрагментацию. Операция выполняется в режиме реального времени, и ее можно прервать в любой момент, не потеряв уже достигнутых результатов. Чтобы реорганизовывать индексы, применяется команда `ALTER INDEX REORGANIZE`.

Перестроение индекса

Этот метод создает еще одну копию индекса в таблице. Это автономная операция, которая блокирует таблицу в версиях SQL Server, отличных от Enterprise. В версии Enterprise она может выполняться в реальном времени, хотя по-прежнему потребуется непродолжительная блокировка на уровне таблицы в начале и конце операции. В этом случае для перестроения индексов применяется команда `ALTER INDEX REBUILD`.

В документации Microsoft[1] рекомендуется перестраивать индексы, если их внешняя фрагментация (`avg_fragmentation_in_percent`) превышает 30 %, и реорганизовывать индексы при фрагментации от 5 до 30 %. Эти значения можно взять за основу, но, вероятно, вам стоит проанализировать свою систему и скорректировать параметры под свои нужды.

SQL Server 2017 и более поздних версий в редакции Enterprise, а также базы данных Azure SQL позволяют приостанавливать и возобновлять создание индексов и операции перестроения в реальном времени. (Возобновляемые операции создания индекса в реальном времени появились в SQL Server 2019.) Это дает больше гибкости, когда вы разрабатываете стратегию обслуживания индекса, и позволяет во время операции усекать журнал транзакций (см. главу 11) за счет выделения дополнительного пространства для хранения данных.

Обратите внимание на свойство индекса `FILLFACTOR`. Оно позволяет резервировать часть свободного пространства во время создания или перестроения индекса, в результате чего впоследствии потребуется меньше разбиений страниц. Рекомендую устанавливать `FILLFACTOR` меньше 100 %, кроме случаев, когда ваш индекс только пополняется и постоянно увеличивается. Я обычно начинаю

[1] https://oreil.ly/61dzF

с 85 или 90 % и оптимизирую значения, чтобы получить минимальную внутреннюю и внешнюю фрагментацию в индексе.

Наконец, для таблиц-куч есть представление `sys.dm_db_index_physical_stats`, которое содержит информацию об указателях переадресации в столбце `forwarded_record_count`. Таблицы с большим количеством таких указателей неэффективны, и их можно перестроить с помощью операции `ALTER TABLE REBUILD`. Однако в большинстве случаев лучше преобразовать их в таблицы с кластеризованным индексом. Подробнее о том, как обнаруживать неэффективные таблицы-кучи, я расскажу в главе 14.

Ола Халленгрен (Ola Hallengren) поддерживает библиотеку сценариев[1], которые фактически стали стандартом для задач обслуживания баз данных. Возможно, вам тоже стоит использовать их в своих системах.

Статистика и оценка количества элементов

Информацию о распределении данных в индексе SQL сервер хранит во внутренних *объектах статистики*. По умолчанию SQL Server создает статистику для каждого индекса в базе данных и использует ее при оптимизации запросов. Давайте сначала посмотрим, какая информация хранится в статистике.

Листинг 5.2 создает таблицу с кластеризованными и некластеризованными индексами и заполняет ее тестовыми данными. Затем с помощью команды `DBCC SHOW_STATISTICS` вызывается информация о статистике.

Листинг 5.2. Пример вывода статистики

```
CREATE TABLE dbo.DBObjects
(
    ID INT NOT NULL IDENTITY(1,1),
    Name SYSNAME NOT NULL,
    CreateDate DATETIME NOT NULL
);

CREATE UNIQUE CLUSTERED INDEX IDX_DBObjects_ID
ON dbo.DBObjects(ID);

INSERT INTO dbo.DBObjects(Name,CreateDate)
    SELECT name, create_date FROM sys.objects ORDER BY name;

-- Создать несколько одинаковых значений
INSERT INTO dbo.DBObjects(Name, CreateDate)
    SELECT t1.Name, t1.CreateDate
```

[1] https://oreil.ly/hz0R2

```
    FROM dbo.DBObjects t1 CROSS JOIN dbo.DBObjects t2
    WHERE t1.ID = 5 AND t2.ID between 1 AND 20;

CREATE NONCLUSTERED INDEX IDX_DBObjects_Name_CreateDate
ON dbo.DBObjects(Name, CreateDate);

DBCC SHOW_STATISTICS('dbo.DBObjects','IDX_DBObjects_Name_CreateDate');
```

На рис. 5.11 показан вывод этого кода (в вашей системе результаты могут отличаться).

	Name	Updated	Rows	Rows Sampled	Steps	Density
1	IDX_DBObjects_Name_CreateDate	Apr 23 2022 9:36AM	136	136	111	1

Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
37.27941	YES	*NULL*	136	0

	All density	Average Length	Columns
1	0.00862069	25.279411	Name
2	0.00862069	33.27941	Name, CreateDate
3	0.007352941	37.27941	Name, CreateDate, ID

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	_trusted_assemblies	0	1	0	1
2	EventNotificationErrorsQueue	0	1	0	1
3	external_library_setup_failur	0	1	0	1
4	MSreplication_options	0	1	0	1
5	P1	0	21	0	1
6	P2	0	1	0	1
7	P2a	0	1	0	1
8	P3	0	1	0	1
9	persistent_version_store	0	1	0	1

Рис. 5.11. Статистика

Как видите, команда `DBCC SHOW_STATISTICS` возвращает три набора результатов. Первый содержит метаданные о статистике, такие как имя, дата обновления и количество строк в индексе на момент обновления статистики.

Второй набор, называемый *вектором плотности*, содержит информацию о плотности для комбинации ключевых значений статистики (индекса). Он рассчитывается по формуле `1 / количество различных значений` и показывает, сколько строк в среднем соответствуют каждой комбинации ключевых значений. Обратите внимание, что хотя индекс `IDX_DBObjects_Name_CreateDate` имеет два

ключа, идентификатор строки `ID` (столбец кластеризованного индекса) тоже присутствует в индексе и возвращается в составе вектора плотности.

Последний и самый важный набор результатов называется *гистограммой*, которую также можно получить через представление `sys.dm_db_stats_histogram`[1]. Это информация о распределении данных в индексе. В каждую запись в гистограмме, называемую *шагом гистограммы*, входят выборочное ключевое значение из крайнего левого столбца статистики (индекса) и информация о распределении данных в интервале от предыдущего до текущего значения `RANGE_HI_KEY`. Шаг гистограммы также содержит оценочное количество строк в интервале (`RANGE_ROWS`), количество строк со значением ключа, равным `RANGE_HI_KEY` (`EQ_ROWS`), количество уникальных значений ключа в интервале (`DISTINCT_RANGE_ROWS`) и среднее количество строк на каждое уникальное значение ключа (`AVG_RANGE_ROWS`).

Во время оптимизации запроса SQL Server использует статистики, чтобы оценить количество строк, которые каждый оператор в плане выполнения должен обработать и вернуть следующему оператору. Этот процесс называется *оценкой количества элементов (cardinality estimation)*.

Оценка количества элементов существенно влияет на план выполнения. SQL Server использует ее, чтобы задать порядок операторов в плане, индексы для доступа к данным, типы операторов соединения и многое другое. Эффективность планов выполнения во многом зависит от того, правильно ли оценено количество элементов, а следовательно, от точности статистики.

Есть три вещи, которые нужно знать о статистике. Первая и самая важная: SQL Server хранит гистограмму и информацию о распределении данных только для крайнего левого столбца индекса, а не для остальных столбцов индекса или комбинаций значений столбцов индекса.

В составных индексах часто предлагают делать самый избирательный столбец крайним левым. Хотя это поможет улучшить оценку количества элементов, не следуйте этому принципу слепо. Проанализируйте запросы и убедитесь, что предикаты в крайних левых столбцах поддерживают SARG, а также эффективные операции поиска по индексу.

Второй важный момент заключается в том, что гистограмма хранит не более 200 шагов независимо от размера таблицы и от того, секционирована ли она. Это может повлиять на оценку количества элементов в больших таблицах с неравномерным распределением данных, потому что каждый шаг содержит информацию о бóльших интервалах ключей.

Наконец, надо понимать, как SQL Server обновляет статистику. В базах данных с уровнем совместимости ниже 130 (начиная с SQL Server 2016) статистика

[1] https://oreil.ly/Tk0tv

обновляется автоматически, *только* если в индексе изменилось 20 % данных. Например, в таблице со 100 миллионами строк вам потребуется вставить, удалить или обновить ключевые столбцы индекса в 20 миллионах строк, прежде чем сработает автоматическое обновление. Это означает, что в больших таблицах статистика редко обновляется автоматически, так что со временем она становится неточной.

Начиная с уровня совместимости 130, действует динамический порог обновления статистики. Чем больше объем данных в таблице, тем меньший процент изменений нужен, чтобы вызвать обновление статистики. Для баз данных с более старыми уровнями совместимости, а также в старых версиях SQL Server можно настроить такое же поведение с помощью флага трассировки `T2371`. Это один из флагов, которые я включаю в каждой системе.

Как вести статистику

Точная и актуальная статистика улучшает производительность системы. Устраняя неполадки, анализируйте стратегию ведения статистики и проверяйте, дает ли она точную информацию.

Можно полагаться на автоматическое обновление статистики, можно управлять статистикой вручную, а можно сочетать оба подхода. Стратегия ведения статистики зависит также от обслуживания индекса, потому что при перестроении индекса статистика в нем автоматически обновляется. Однако она не обновляется при реорганизации индекса.

Чтобы SQL Server автоматически создавал и обновлял статистику на уровне базы данных, используйте параметры базы данных *Auto Create Statictics* и *Auto Update Statistics*. Когда они включены, SQL Server автоматически ведет статистику по всем индексам, кроме тех, для которых активен параметр `STATISTICS_NORECOMPUTE` (по умолчанию он неактивен).

В SQL Server предусмотрено несколько методов обновления статистики. По умолчанию SQL Server просто оценивает значения по выборке данных из индекса. Это простой подход, но его результаты не всегда точны. Противоположный подход — обновлять статистику с помощью инструкции `UPDATE STATISTICS WITH FULLSCAN`, которая прочитает весь индекс.

Также можно обновить статистику, указав процент или количество строк для выборки с помощью инструкции `UPDATE STATISTICS WITH SAMPLE`. Очевидно, что чем больше данных вы прочитаете, тем больше будет нагрузка на ввод/вывод для больших индексов, но и тем точнее будут результаты. Обновление с помощью `FULLSCAN` или больших выборок полезно, если вы можете позволить себе накладные расходы: например, запустить процедуру в период низкой активности.

Во время компиляции запроса SQL Server проверяет, не устарела ли статистика, и может обновить ее синхронно или асинхронно в зависимости от значения параметра базы данных *Update Statistics Asynchronously*. При синхронном обновлении оптимизатор запросов откладывает компиляцию запроса, пока обновление не завершится. При асинхронном обновлении запрос оптимизируется с использованием старой статистики, а сама статистика обновляется в фоновом режиме. Можно оставить синхронное обновление статистики по умолчанию, если только в вашей системе не требуется экстремально короткое время отклика на запросы.

Заданные по умолчанию пороговые значения, при которых запускается автоматическое обновление статистики, часто бывают приемлемы, если база данных имеет уровень совместимости 130 или выше или если установлен флаг `T2371`. Однако в некоторых случаях стоит обновлять статистику ключевых индексов вручную и/или запустить обновление статистики с помощью `FULLSCAN` в нерабочее время.

Статистику по отфильтрованным индексам обычно лучше обновлять вручную. Изменения отфильтрованных столбцов не учитываются с точки зрения порога обновления статистики, и автоматическое ведение статистики из-за этого становится неэффективным. В сопутствующих материалах к книге приведен сценарий, демонстрирующий такую ситуацию.

Отфильтрованные индексы позволяют фильтровать подмножества данных в таблице, что уменьшает размер индекса и затраты на его обслуживание. Подробности можно найти в документации Microsoft[1].

В листинге 5.3 показано, как просмотреть свойства статистики: например, когда она в последний раз обновлялась и сколько изменений в данных произошло с момента последнего обновления.

Листинг 5.3. Анализ свойств статистики

```
SELECT
    s.stats_id AS [Stat ID]
    ,sc.name + '.' + t.name AS [Table]
    ,s.name AS [Statistics]
    ,p.last_updated
    ,p.rows
    ,p.rows_sampled
    ,p.modification_counter AS [Mod Count]
FROM
    sys.stats s JOIN sys.tables t ON
        s.object_id = t.object_id
    JOIN sys.schemas sc ON
        t.schema_id = sc.schema_id
```

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

```
    OUTER APPLY
        sys.dm_db_stats_properties(t.object_id,s.stats_id) p
ORDER BY
    p.last_updated
```

В этом разделе мы лишь слегка коснулись вопросов статистики и ее ведения. Настоятельно рекомендую почитать документацию Microsoft[1], чтобы изучить эту тему глубже.

Модели оценки количества элементов

Вы уже знаете, что качество оптимизации запросов зависит от точности оценки количества элементов. Чтобы составить эффективный план выполнения, SQL Server должен правильно оценить количество строк на каждом шаге выполнения запроса. Точная статистика важна для улучшения оценок, однако это только часть общей картины.

Оценивая количество элементов, оптимизатор запросов опирается на ряд предположений, в том числе по следующим вопросам:

- Как распределены данные в таблицах.
- Как различные операторы и предикаты влияют на размер вывода.
- Как связаны друг с другом различные предикаты в одной и той же таблице.
- Как соотносятся между собой данные в нескольких таблицах во время соединения.

Эти предположения совместно с алгоритмами оценки количества элементов определяют, какая модель будет использована во время оптимизации.

Исходная (устаревшая) модель оценки была первоначально разработана для SQL Server 7.0, и других моделей не существовало до версии SQL Server 2014. Если не считать незначительных улучшений от версии к версии, все это время модель в целом оставалась одной и той же.

В SQL Server 2014 корпорация Microsoft внедрила новую модель оценки количества элементов, доступную в базах данных с уровнем совместимости 120. В этой модели используются другие предположения, и в результате получаются другие оценки количества элементов и планы выполнения.

Нельзя точно сказать, какая модель «правильнее». Одни запросы лучше работают с новой моделью, другие — со старой. В новых версиях SQL Server можно продолжать использовать устаревшую модель оценки количества элементов, однако в какой-то момент все-таки будет лучше обновиться. Microsoft заявля-

[1] https://oreil.ly/w268o

ет, что не собирается в будущем удалять устаревшую модель из SQL Server, но и улучшать ее не будет.

К сожалению, перейти на новую модель — это легче сказать, чем сделать, особенно в больших и сложных системах. Смена модели может привести к масштабным изменениям в планах выполнения, поэтому вы должны быть готовы быстро находить и устранять возникшие регрессы. К счастью, хранилище запросов может упростить переход. Можно собрать данные перед сменой модели и заставить SQL Server использовать старые планы выполнения для тех запросов, которые регрессировали в рамках новой модели. Очевидно, впоследствии их все равно придется проанализировать и оптимизировать.

Моделью оценки количества элементов можно управлять с помощью уровня совместимости базы данных. На каждом уровне совместимости, начиная со 120 (SQL Server 2014), новая модель может вести себя немного по-разному. В то же время устаревшие модели ведут себя одинаково в любой версии SQL Server. На оценку также может повлиять параметр базы данных `QUERY_OPTIMIZER_HOTFIXES` или флаг трассировки `T4199` (и то и другое обсуждалось в главе 1).

В SQL Server 2014 можно управлять моделью с помощью как уровней совместимости базы данных, так и флагов трассировки. Флаги `T2312` и `T9481` заставляют SQL Server использовать новую и устаревшую модели соответственно, игнорируя уровень совместимости базы данных. В SQL Server 2016 и более поздних версиях можно продолжать использовать старую модель с новыми уровнями совместимости, установив параметр базы данных `LEGACY_CARDINALITY_ESTIMATION`.

Выполнять обновление версии SQL Server лучше поэтапно, чтобы снизить риск регресса. Сперва обновите версию сервера, оставив старую модель оценки количества элементов. Убедитесь, что после обновления все работает корректно. Тогда можно подумать о смене модели. Как уже упоминалось, в этой процедуре можно использовать хранилище запросов.

Как правило, я не рекомендую переходить на новую модель оценки количества элементов в SQL Server 2014. В ранних сборках этой версии я столкнулся с ошибками, которые привели к регрессу запросов. Если вы все-таки меняете модель, то установите последний пакет обновлений, включите флаг `T4199` и тщательно протестируйте систему. Также стоит отметить, что в SQL Server 2014 нет хранилища запросов, что значительно усложняет переход на новую модель.

Наконец, в базах данных с уровнем совместимости 160 (SQL Server 2022) и выше доступна новая функция интеллектуальной обработки запросов — *обратная связь оценки количества элементов (cardinality estimation feedback)*. Когда SQL Server во время выполнения запроса обнаруживает существенные ошибки оценки количества элементов, он может перекомпилировать запрос, используя другие предположения модели. Затем SQL Server проверяет, стало ли лучше, и в зависимости от этого либо сохраняет, либо отбрасывает новый план.

Обратная связь оценки количества элементов опирается на указания хранилища запросов и требует, чтобы оно было включено. Более того, SQL Server использует обратную связь не для каждого запроса, а только если ошибка оценки значительна, а запрос выполняется достаточно часто. В любом случае имеет смысл переключить базы данных на последний уровень совместимости, потому что другие функции интеллектуальной обработки данных вам все равно пригодятся, даже если оставить устаревшую оценку количества элементов.

Анализ плана выполнения

Когда SQL Server оптимизирует запрос, он создает план выполнения запроса. Этот план состоит из множества операторов, которые получают доступ к данным и манипулируют ими, чтобы выдать результат запроса. Занимаясь тонкой настройкой запросов, нужно анализировать и улучшать их планы выполнения. Хотя любой специалист по базам данных знаком с планами выполнения, тем не менее я хотел бы обсудить несколько моментов, касающихся настройки запросов. Для начала посмотрим, как SQL Server выполняет операторы в плане.

Построчный и пакетный режим выполнения

У SQL Server есть два режима обработки запросов. *Построчный режим* (по умолчанию) традиционно используется с хранилищами на основе строк и индексами на основе B-деревьев. В этом режиме каждый оператор в плане выполнения обрабатывает строки данных одну за другой, запрашивая их у дочерних операторов по мере необходимости.

Рассмотрим простой запрос, показанный в листинге 5.4.

Листинг 5.4. Построчный режим выполнения: пример запроса

```
SELECT TOP 10 c.CustomerId, c.Name, a.Street, a.City, a.State, a.ZipCode
FROM
    dbo.Customers c JOIN dbo.Addresses a ON
        c.PrimaryAddressId = a.AddressId
ORDER BY
    c.Name
```

Этот запрос порождает план выполнения, показанный на рис. 5.12. SQL Server выбирает все данные из таблицы `Customers`, сортирует их по столбцу `Name`, получает первые 10 строк, объединяет их с данными `Addresses` и возвращает клиенту.

Давайте проанализируем, как SQL Server выполняет запрос. Оператор *Select*, который является родительским оператором в плане выполнения, вызывает метод `GetRow()` оператора *Top*. Оператор *Top*, в свою очередь, вызывает метод `GetRow()` оператора *Nested Loop Join*.

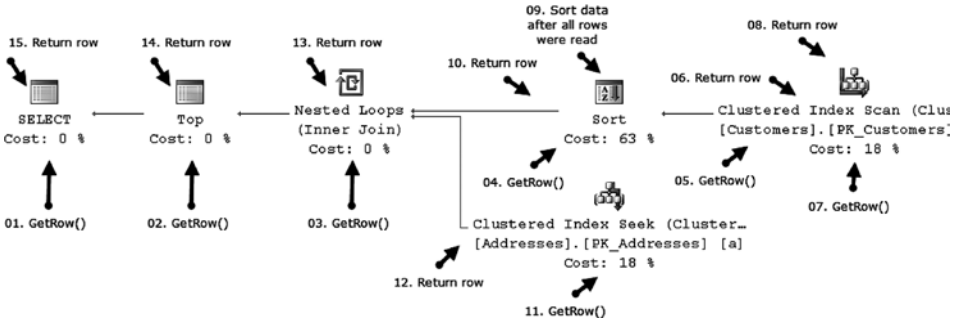


Рис. 5.12. Построчный режим выполнения: получение первой строки

Оператор *Join* получает данные из двух разных источников. Прежде всего он вызывает метод `GetRow()` оператора *Sort*. Чтобы выполнить сортировку, SQL Server должен сначала прочитать все строки. Поэтому операция *Sort* несколько раз вызывает метод `GetRow()` оператора *Clustered Index Scan* (Просмотр кластеризованного индекса), накапливая результаты. Оператор *Scan*, который находится на самом нижнем уровне в дереве плана выполнения, при каждом вызове возвращает по одной строке из таблицы `Customers`. На рис. 5.12 для простоты показаны только два вызова `GetRow()`.

Когда все данные из таблицы `Customers` прочитаны, оператор *Sort* сортирует их и возвращает первую строку оператору *Join*, который после этого вызывает метод `GetRow()` оператора *Clustered Index Seek* (Поиск кластеризованного индекса) для таблицы `Addresses`. Если обнаружено совпадение, то оператор *Join* объединяет данные из обеих таблиц и передает результирующую строку обратно оператору *Top*, который, в свою очередь, передает ее оператору *Select*.

Оператор *Select* возвращает строку клиенту и снова вызывает метод `GetRow()` оператора *Top*, чтобы запросить следующую строку. Процесс повторяется до тех пор, пока не будут выбраны первые 10 строк. Все операторы сохраняют свое состояние, а оператор *Sort* запоминает отсортированные данные. Ему не нужно снова обращаться к оператору *Clustered Index Scan*, как показано на рис. 5.13.

У каждого оператора в плане выполнения есть несколько свойств, имена которых могут немного различаться в разных версиях SSMS и в других приложениях. Вот наиболее важные из них:

Actual Number of Rows и Number of Rows Read

Показывают, сколько строк было возвращено оператором и сколько строк было обработано во время выполнения. Например, оператор *Index Scan* с предикатом может обработать 1000 строк, отфильтровав из них 950. В этом случае в свойствах будет отображаться 50 и 1000 строк соответственно.

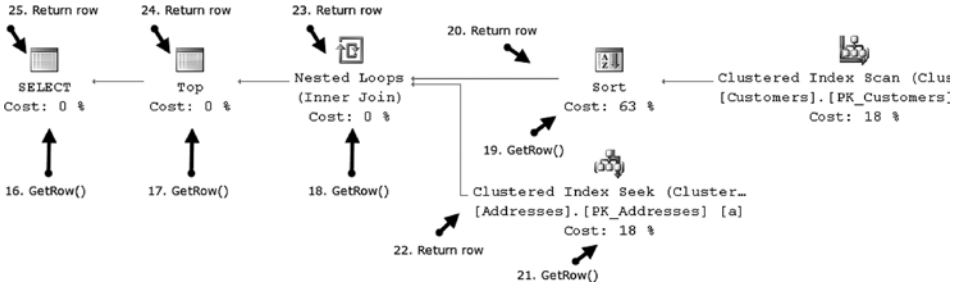


Рис. 5.13. Построчный режим выполнения: получение следующей строки

Estimated Number of Rows и Estimated Number of Rows Read

Содержат результат оценки количества элементов и предполагаемое количество строк (по оценке оптимизатора запросов), которое оператор должен вернуть и обработать. Если предполагаемые и фактические значения существенно расходятся, это указывает на ошибочную оценку количества элементов, из-за которой план выполнения может получиться неоптимальным.

Number of Executions и Estimated Number of Executions

Number of Executions показывает, сколько раз был выполнен оператор. Это не количество вызовов `GetRow()`, а значение, показывающее, сколько раз была обработана эта часть плана выполнения. Например, в плане, показанном на рис. 5.12 и 5.13, оператор *Clustered Index Scan* в таблице `Customers` выполняется один раз, а *Clustered Index Seek* в таблице `Addresses` — 10 раз.

Estimated Number of Executions показывает оценочное количество выполнений, на которое опирается оптимизатор запросов.

Startup Predicate

В некоторых случаях у операторов может быть свойство Startup Predicate — условие, необходимое для выполнения оператора. Например, предложение `WHERE @ProvideDetails = 1` может породить оператор *Filter*, в котором у свойства Startup Predicate будет значение `@ProvideDetails = 1`. Поддерево плана выполнения после оператора *Filter* может выполняться или не выполняться в зависимости от того, как ведет себя параметр `@ProvideDetails` во время выполнения.

К сожалению, построчный режим выполнения и построчная обработка плохо масштабируются в случае больших аналитических запросов, которые работают с миллионами или даже миллиардами строк. Чтобы решить эту проблему, в SQL Server 2012 была представлена другая модель — *выполнение в пакетном режиме*. В этой модели операторы в планах выполнения могут обрабатывать

строки пакетами. Процесс оптимизирован для больших объемов данных и параллельных планов выполнения.

До SQL Server 2019 оптимизатор запросов активировал выполнение в пакетном режиме только тогда, когда хотя бы одна из таблиц в запросе имела индекс columnstore. Это ограничение было снято в Enterprise Edition SQL Server 2019, где пакетный режим можно использовать с индексами B-деревьев на основе строк в базах данных с уровнем совместимости 150. Это не означает, что все планы выполнения станут работать в пакетном режиме, однако оптимизатор запросов будет применять его, когда нужно.

Как и любая функциональность, влияющая на планы выполнения, пакетный режим может вызвать регресс. Этот режим можно включать и отключать на уровне базы данных с помощью параметра `BATCH_MODE_ON_ROWSTORE` или на уровне запроса с помощью указаний `ALLOW_BATCH_MODE` и `DISALLOW_BATCH_MODE`.

Наконец, есть хитрость, которая может активировать пакетный режим для таблиц на основе B-деревьев в SQL Server 2016 и 2017: если такие таблицы используются для больших аналитических запросов, в них можно создать пустые и отфильтрованные некластеризованные индексы columnstore. Например, если у таблицы есть столбец `ID`, содержащий только положительные значения, такой индекс даст оптимизатору запросов доступ к пакетному режиму: `CREATE NONCLUSTERED COLUMNSTORE INDEX NCCI ON T WHERE ID < 0`.

Посмотреть режим выполнения оператора можно с помощью свойства *Actual Execution Mode* в плане выполнения. *Actual Number of Batches* покажет, сколько пакетов было обработано. Тем не менее стратегия настройки запросов будет одинаковой независимо от режима выполнения.

Динамическая статистика запросов и профилирование статистики выполнения

Анализировать планы выполнения можно с помощью различных инструментов. Помимо SSMS, существует еще один бесплатный инструмент от Microsoft — Azure Data Studio[1]. Несмотря на название[2], он отлично работает с локальными экземплярами SQL Server, причем не только на Windows.

Я считаю, что Azure Data Studio предназначена скорее для разработчиков, чем для администраторов баз данных. Тем не менее в ней есть базовые функции администрирования и настройки базы данных, а также возможность подключать

[1] https://oreil.ly/We7gO

[2] Azure — «лазурный», цвет неба. То есть лазурь служит как бы метафорой облачных технологий, а в тексте при этом говорится о том, что Azure работает не только в облаке, но и с локальными экземплярами. — *Примеч. ред.*

сторонние расширения. Некоторые из них даже обеспечивают поддержку других СУБД, помимо SQL Server.

На мой взгляд, лучший бесплатный инструмент для настройки запросов — это SolarWinds Plan Explorer[1] (ранее известный как SentryOne Plan Explorer). Он фокусируется на анализе плана выполнения и кажется мне более продвинутым и простым в использовании, чем SSMS. Советую скачать и протестировать его, если вы этого еще не сделали.

В SSMS есть еще одна полезная функция — *динамическая статистика запросов (Live Query Statistics)*. Она позволяет отслеживать, как выполняется запрос в реальном времени, и обнаруживает возможные дефекты в плане выполнения.

На рис. 5.14 показан пример окна динамической статистики запросов в SSMS (снимок взят из документации Microsoft[2]). Операторы со сплошными линиями выполнены. Пунктирные линии обозначают дерево операторов, которые выполняются в данный момент. Отображается оценочный процент выполнения каждого активного оператора, а также фактическое и оценочное количество строк. Все метрики непрерывно обновляются во время выполнения запроса.

Динамическая статистика очень полезна, когда нужно отлаживать длительные запросы. Она позволяет выявить неэффективные планы выполнения и ускорить дальнейшую настройку запросов. Можно включить динамическую статистику для запросов, которые вы запускаете в SSMS. Доступ к ней также можно получить в разделе Active Expensive Query в окне монитора активности.

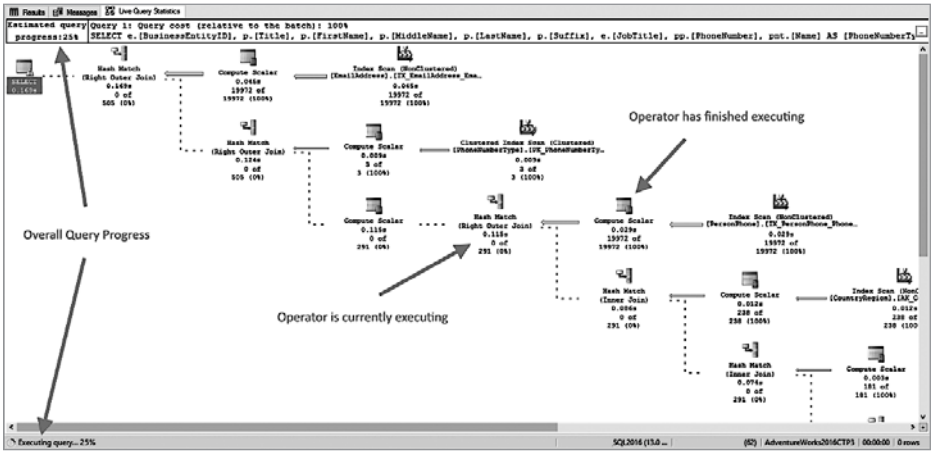


Рис. 5.14. Динамическая статистика запросов (источник: документация Microsoft[3])

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

[2] https://oreil.ly/qUJOV

[3] https://oreil.ly/Rpy4O

Динамическая статистика запросов собирает данные на основе *профилирования статистики выполнения запросов*. SQL Server может собирать данные двумя методами. Исторически, чтобы получить фактический план выполнения запросов, во всех версиях SQL Server использовалось *стандартное профилирование*. К сожалению, это очень ресурсоемкий метод.

Начиная с SQL Server 2014 с пакетом обновления 2 (SP2), существует еще один метод, называемый *упрощенным профилированием*. У него значительно меньше накладных расходов, но он не собирает информацию о ЦП во время выполнения.

В табл. 5.2 показано, как включить профилирование в разных версиях SQL Server. Здесь же перечислены расширенные события, которые обеспечивают глобальное профилирование в системе. Динамическая статистика запросов интегрируется с последней версией профилирования, поддерживаемой экземпляром SQL Server, на котором она выполняется.

Таблица 5.2. Профилирование статистики выполнения запросов

	Тип	Как включить	Расширенное событие
Версии до SQL Server 2014 SP2	Стандартное	`SET STATISTICS XML` `SET STATISTICS PROFILE`	`query_post_execution_showplan`
SQL Server 2014 SP2–SQL Server 2016 RTM	Упрощенное, версия 1	Динамическая статистика запросов	`query_post_execution_showplan` (меньше накладных расходов по сравнению с прежними версиями SQL Server)
SQL Server 2016 SP1–SQL Server 2017	Упрощенное, версия 2	`T7412` Подсказка `QUERY_PLAN_PROFILE`	`query_plan_profile` Только для операторов с указанием `QUERY_PLAN_PROFILE`. Полезно, если нужно захватить план одного запроса из большого пакета модуля T-SQL
SQL Server 2019	Упрощенное, версия 3	Включено по умолчанию параметром БД `LIGHTWEIGHT_QUERY_PROFILING`	`query_post_execution_plan_profile`

Накладные расходы у стандартного профилирования весьма велики, а при упрощенном профилировании они ниже. По данным Microsoft, начиная с SQL Server 2016 с пакетом обновления 1 (SP1), добавочная нагрузка непрерывно работающего упрощенного профилирования составляет от 2 до 4 %, хотя она варьируется и для некоторых рабочих нагрузок может оказаться значительно выше. Будьте осторожны и измеряйте, как профилирование влияет на вашу систему, особенно если вы планируете использовать его долговременно.

Есть еще одна полезная новая функция — `sys.dm_exec_query_statistics_xml`[1], которая использует упрощенное профилирование (и возвращает `NULL`, если оно выключено). Эта функция возвращает оперативный план выполнения текущего запроса. Результат выглядит примерно так же, как снимок динамической статистики запросов. Функцию можно использовать вместе с представлением `sys.dm_exec_requests`, как показано в листинге 5.5.

Листинг 5.5. Использование `sys.dm_exec_query_statistics_xml`

```
SELECT
    er.session_id
    ,er.request_id
    ,DB_NAME(er.database_id) as [database]
    ,er.start_time
    ,CONVERT(DECIMAL(21,3),er.total_elapsed_time / 1000.) AS [duration]
    ,er.cpu_time
    ,SUBSTRING(
        qt.text,
        (er.statement_start_offset / 2) + 1,
        ((CASE er.statement_end_offset
            WHEN -1 THEN DATALENGTH(qt.text)
            ELSE er.statement_end_offset
        END – er.statement_start_offset) / 2) + 1
    ) AS [statement]
    ,er.status
    ,er.wait_type
    ,er.wait_time
    ,er.wait_resource
    ,er.blocking_session_id
    ,er.last_wait_type
    ,er.reads
    ,er.logical_reads
    ,er.writes
    ,er.granted_query_memory
    ,er.dop
    ,er.row_count
    ,er.percent_complete
    ,es.login_time
    ,es.original_login_name
    ,es.host_name
    ,es.program_name
    ,c.client_net_address
    ,ib.event_info AS [buffer]
    ,qt.text AS [sql]
    ,p.query_plan
FROM
    sys.dm_exec_requests er WITH (NOLOCK)
        OUTER APPLY sys.dm_exec_input_buffer(er.session_id, er.request_id) ib
        OUTER APPLY sys.dm_exec_sql_text(er.sql_handle) qt
```

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

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

Характерные проблемы при настройке запросов

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

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

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

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

Неэффективный код

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

Предикаты, не поддерживающие поиск и аргументы (SARG)

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

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

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

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

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

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

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

Пользовательские функции

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

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

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

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

Временные таблицы и табличные переменные

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

С временными таблицами и табличными переменными связаны две распространенные ошибки.

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

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

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

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

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

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

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

Хранимые процедуры и ORM-фреймворки

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

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

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

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

Неэффективный поиск по индексу

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

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

Листинг 5.6. Неэффективность поиска по индексу

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

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

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

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

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

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

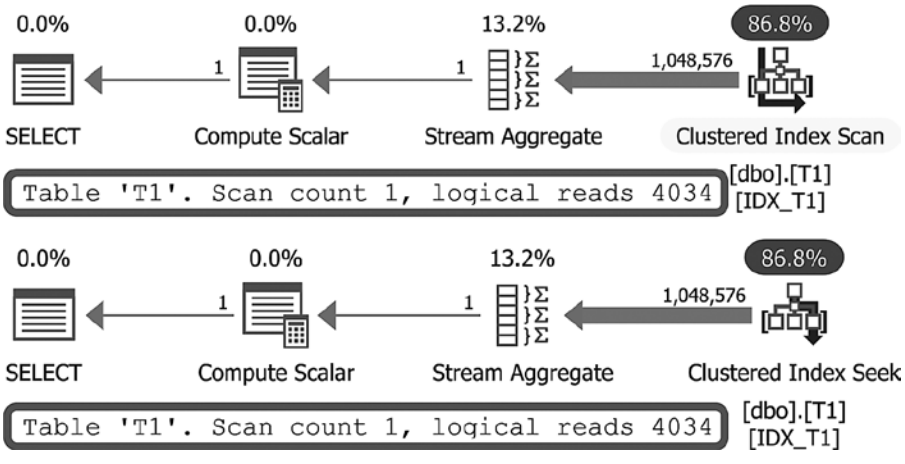


Рис. 5.15. Неэффективный поиск по индексу

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

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

Листинг 5.7. Пример запроса

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

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

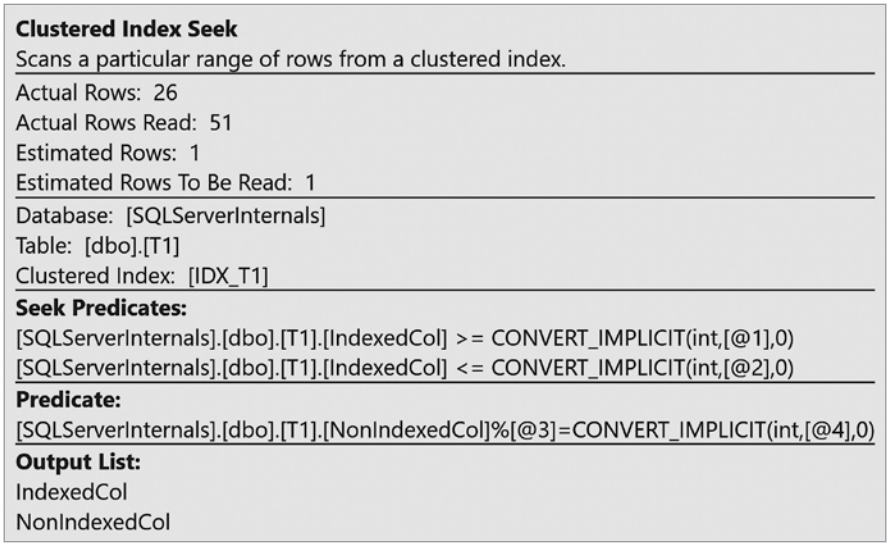


Рис. 5.16. Свойства поиска по индексу

Рассмотрим самые важные свойства оператора *Index Seek*.

Seek predicate (Предикат поиска)

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

Predicate

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

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

Actual rows и Actual rows read

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

Estimated rows и Estimated rows to be read

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

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

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

Неправильный тип соединения

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

Давайте рассмотрим все три типа подробнее.

Соединение в цикле

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

Листинг 5.8. Алгоритм соединения в цикле (псевдокод)

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

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

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

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

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

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

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

Соединение слиянием

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

Листинг 5.9. Алгоритм соединения слиянием (псевдокод)

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

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

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

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

Хеш-соединение

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

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

Листинг 5.10. Алгоритм внутреннего хеш-соединения (псевдокод)

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

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

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

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

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

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

Сравнение типов соединений

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

Таблица 5.4. Сравнение типов соединений

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

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

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

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

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

Листинг 5.11. Эффективность соединения: создание таблицы

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

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

Листинг. 5.12. Эффективность производительности: тестовые примеры

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

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

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

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

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

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

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

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

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

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

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

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

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

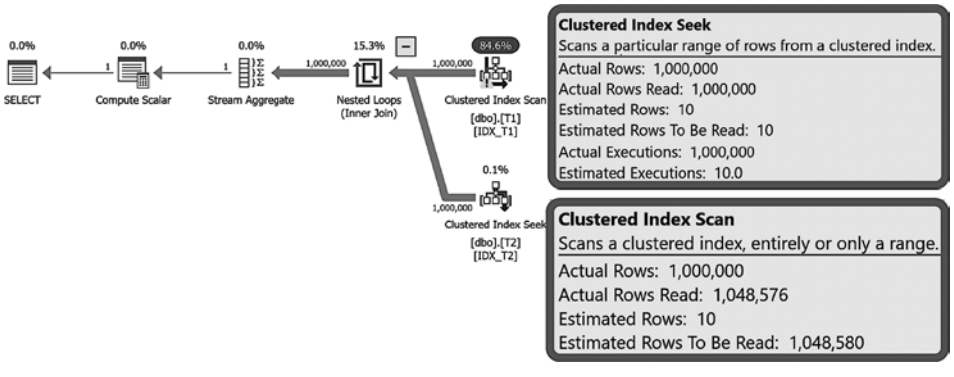


Рис. 5.17. Ошибка оценки количества элементов при соединении в цикле

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

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

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

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

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

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

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

Листинг 5.13. Принудительные типы соединений

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

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

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

Избыточный поиск по ключу

Как вы уже знаете, поиск по ключу и по идентификатору строк[1] становится крайне неэффективным в больших масштабах. SQL Server не использует индексы для операций поиска, когда оценивает, что понадобится большое количество операций поиска по ключу. Однако вы все равно можете столкнуться с избыточным поиском в планах выполнения.

Такая ситуация часто возникает из-за неправильной оценки количества элементов. Если SQL Server рассчитывает, что потребуется всего несколько операций поиска по ключу, он может использовать поиск по некластеризованному индексу, и в случае ошибочной оценки произойдет очень много операций поиска по ключу. Такая ошибка обычно возникает из-за ограничений модели оценки количества элементов или сканирования параметров (в планах, чувствительных к параметрам, о которых пойдет речь в следующей главе).

Эту проблему можно выявить, сравнив ожидаемое и фактическое количество строк в плане выполнения, как показано на рис. 5.18.

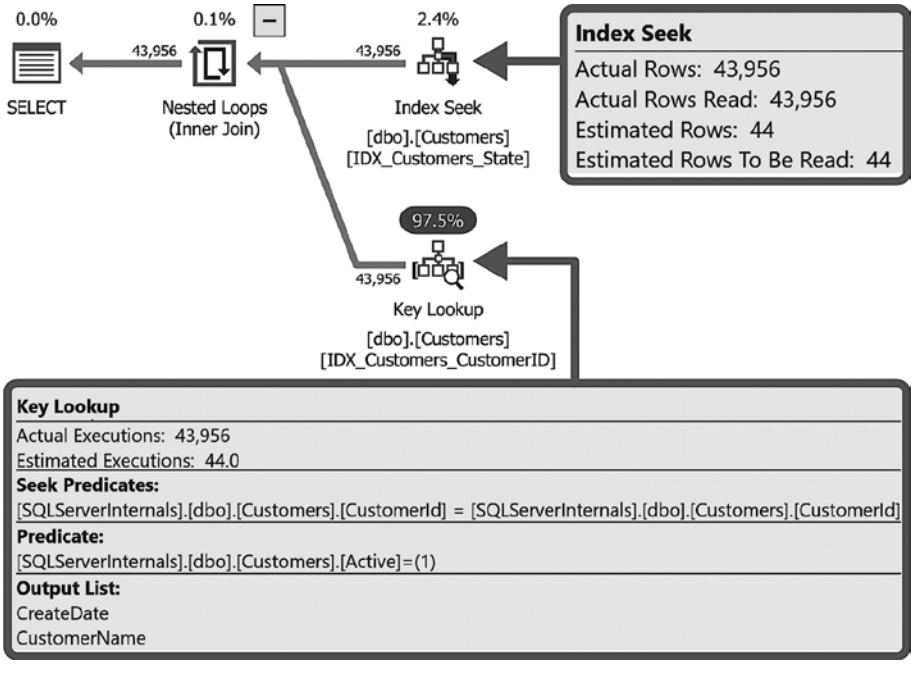


Рис. 5.18. Неправильная оценка количества элементов и поиск по ключу

[1] Для краткости с этого момента я перестану говорить «поиск по ключу и идентификатору строк». Все, что говорится о «поиске по ключу», будет применимо и к поиску по идентификатору строк.

Также SQL Server может прибегнуть к поиску по ключу, выбирая между плохой и наихудшей стратегиями доступа к данным. Запускать миллионы операций поиска по ключу крайне неэффективно, но это все равно лучше, чем просматривать таблицу с миллиардами строк.

Часто можно избавиться от поиска по ключу, используя покрывающие индексы. Если все столбцы, необходимые для запроса, находятся в некластеризованном индексе, SQL Server не требуется доступ к основной строке данных в кластеризованном индексе или куче. По определению такой некластеризованный индекс содержит все столбцы в ключе некластеризованного индекса, а столбцы кластеризованного индекса — в идентификаторе строки.

SQL Server позволяет включать в индекс другие столбцы с помощью предложения `INCLUDE`. Данные из этих столбцов хранятся только на листовом уровне. Они не добавляются к ключу индекса и не влияют на порядок сортировки строк индекса. На рис. 5.19 показана структура индекса с неключевыми столбцами, определенного как `CREATE INDEX IDX_Customers_Name ON dbo.Customers(Name) INCLUDE(DateOfBirth)` в таблице, где `CustomerId` — это кластеризованный индекс.

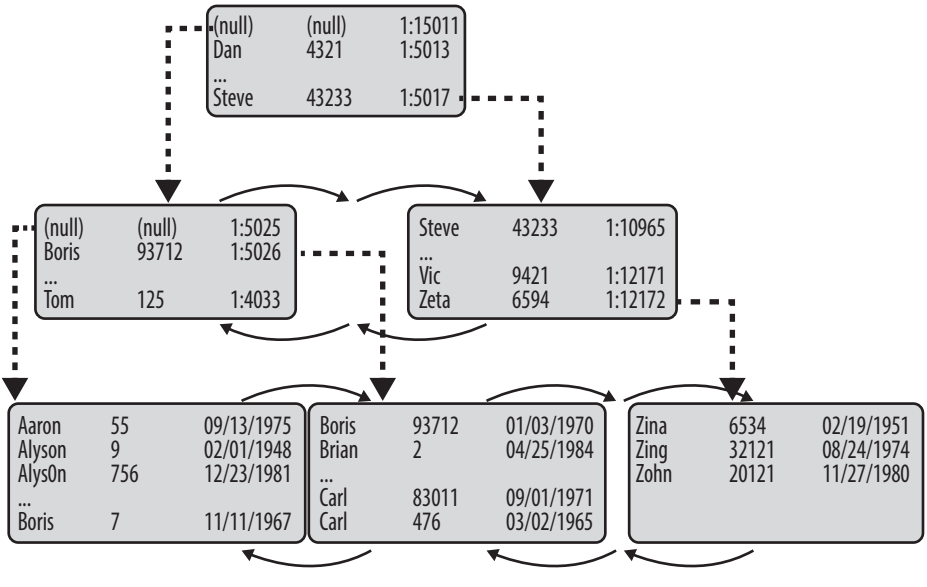


Рис. 5.19. Индекс с неключевыми столбцами

Теперь, если в индексе присутствуют все столбцы, на которые ссылается запрос, SQL Server может получить все данные с листового уровня B-дерева некластеризованного индекса, не выполняя поиск по ключу. Индекс может использоваться независимо от того, сколько строк из него будет выбрано.

Превращение некластеризованных индексов в покрывающие — один из самых популярных методов оптимизации запросов. Можно открыть свойства оператора *Key Lookup* (Поиск ключа) в плане выполнения (см. рис. 5.18) и получить список столбцов из вывода оператора и предиката фильтра. Если включить эти столбцы в некластеризованный индекс, то поиск не понадобится.

Покрывающие индексы — отличный инструмент для оптимизации запросов, но у них есть своя цена. Каждый столбец в индексе увеличивает размер строки в нем, а также количество страниц данных, которые индекс занимает на диске и в памяти. Это приводит к дополнительным накладным расходам при обслуживании индекса и увеличивает размер базы данных. Кроме того, запросам приходится считывать больше страниц при просмотре всего индекса или его части. Это не обязательно влияет на производительность при просмотре небольшого диапазона (в этом случае намного эффективнее прочесть несколько дополнительных страниц, чем выполнять поиск по ключу), однако может ухудшить производительность запросов, которые просматривают много страниц данных или весь индекс.

Покрывающие индексы также вызывают накладные расходы на обновление. Добавляя столбец в некластеризованный индекс, вы храните данные в нескольких местах. Это повышает производительность запросов, выбирающих данные. Но во время обновлений SQL Server вынужден изменять строки в каждом индексе, где оказались обновленные столбцы.

Не рекомендуется создавать очень широкие некластеризованные индексы, включающие большинство столбцов таблицы. Слишком много индексов — тоже плохая идея. В обоих случаях размер базы данных увеличится, и возникнут избыточные затраты на обновление, особенно в средах OLTP. (Подробнее об анализе индексов и консолидации я расскажу в главе 14.)

Наконец, я хотел бы отметить одну очень важную и очевидную вещь. Хотя избыточный поиск по ключу плохо влияет на производительность, сам по себе поиск по ключу — это *абсолютно нормально*. Поиск по сотням или даже тысячам строк — это обычно лучше, чем огромные покрывающие индексы.

Взгляните на это с другой точки зрения: поиск по ключу — это соединение в цикле с эффективным поиском по индексу во внутренней таблице (кластеризованном индексе). В небольших масштабах это очень быстрая и эффективная операция.

Никто не заставляет вас исключать из плана выполнения все поиски по ключу. Проанализируйте их эффективность и избавьтесь только от неэффективных поисков.

Индексирование данных

Разработка правильных индексов — это и искусство, и наука. Со временем вы овладеете этим навыком. Позвольте мне дать несколько советов о том, с чего начать.

У большинства запросов в системе есть параметры. Самые избирательные предикаты с поддержкой SARG отфильтровывают бóльшую часть данных, так что столбцы в этих предикатах — лучшие кандидаты на роль индекса.

Давайте рассмотрим гипотетический запрос, который возвращает список заказов одного клиента (листинг 5.14).

Листинг 5.14. Выбор списка заказов клиента

```
SELECT c.CustomerName, c.CustomerNumber, o.OrderId, o.OrderDate, o.Amount
FROM
    dbo.Customers c JOIN dbo.Orders o ON
        c.CustomerId = o.CustomerId
WHERE
    c.CustomerNumber = @CustNum AND
    c.Active = 1 AND
    o.OrderDate between @StartDate AND @EndDate AND
    o.Fulfilled = 1;
```

Предположим, вы запускаете этот запрос для таблиц, у которых нет индексов, помимо кластеризованных индексов в столбцах `CustomerId` и `OrderId`. План выполнения запроса состоит из двух просмотров кластеризованного индекса и хеш-соединения между таблицами (рис. 5.20). Как видите, это неэффективно.

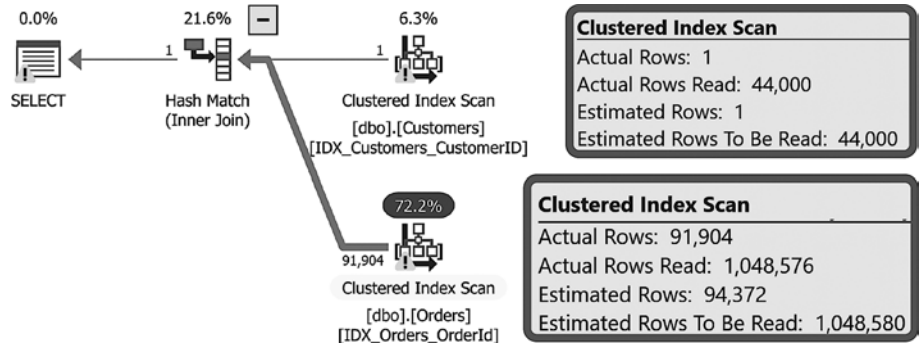


Рис. 5.20. Пример индексации: первоначальный план выполнения

Более естественный подход — начать со столбца `CustomerNumber`. Данные в нем, вероятно, будут уникальными, так что индекс столбца окажется очень избирательным. Предикат `Active=1` проверяет статус клиента. Можно ожидать, что он будет широко использоваться в запросах. Рекомендуется добавить столбец `Active` в индекс с помощью `INCLUDE`.

Можно также ожидать, что `CustomerName` будет часто выбираться вместе с `CustomerNumber` и добавляться в индекс в качестве еще одного неключевого столбца, так что поиск по ключу не понадобится. Тем не менее поиск по ключу

может быть вполне приемлемым в небольшом масштабе с очень избирательными индексами.

На рис. 5.21 показан план выполнения после создания следующего индекса: `CREATE UNIQUE INDEX IDX_Customers_CustomerNumber ON dbo.Customers(CustomerNumber) INCLUDE (Active, CustomerName)`.

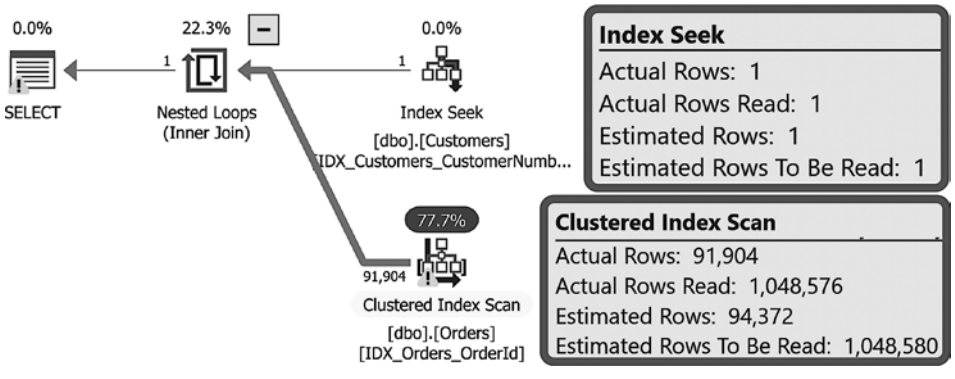


Рис. 5.21. Пример индексации: план с индексом в таблице `Customer`

Столбцы таблицы `Orders` задействованы в трех предикатах. Два из этих предикатов — со столбцами `CustomerId` и `OrderDate` — являются избирательными и поэтому хорошо годятся на роль индекса. Столбцы в индексе могут идти в разном порядке: `(CustomerId, OrderDate)` или `(OrderDate, CustomerId)`.

Чтобы выбрать правильный порядок, посмотрите, как данные будут отсортированы в индексе. В первом варианте `(CustomerId, OrderDate)` SQL Server сначала сортирует данные по `CustomerId`. Затем заказы для каждого клиента сортируются по `OrderDate`. Во втором варианте данные будут отсортированы по `OrderDate` по всем клиентам.

Оба индекса позволяют искать по индексу в таблице `Orders`. Однако первый индекс более эффективен для нашего запроса. SQL Server сможет выполнять просмотр диапазона для заказов, принадлежащих этому единственному клиенту, за интервал времени от `@StartDate` до `@EndDate`. Если же использовать второй индекс, то SQL Server должен будет извлечь все заказы за этот временной интервал для всех клиентов, а для этого придется просматривать больше данных.

Можно добавить в индекс неключевой столбец `Fulfilled`, чтобы вычислять предикат в составе операции поиска. В этом случае я бы также добавил столбец `Amount`. Он достаточно мал и несильно увеличит размер индекса.

На рис. 5.22 показан окончательный план выполнения после создания следующего индекса:

```
CREATE INDEX IDX_Orders_CustomerId_OrderDate ON dbo.Orders(CustomerId,
OrderDate) INCLUDE (Fulfilled, Amount).
```

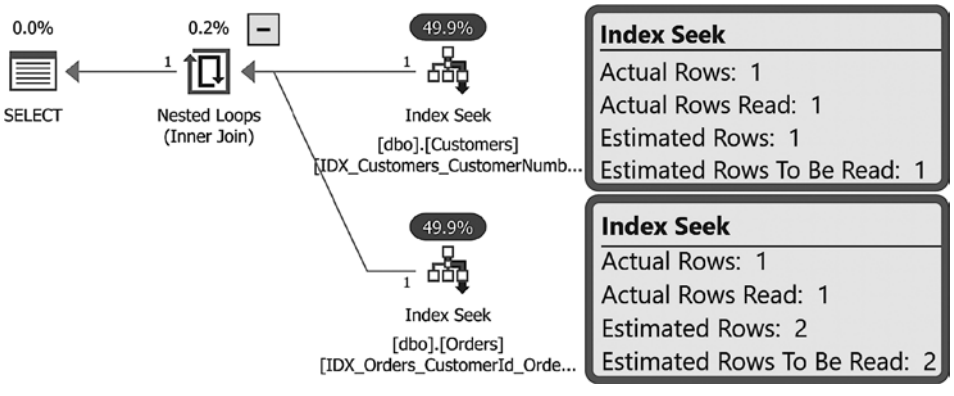


Рис. 5.22. Пример индексации: окончательный план выполнения

С оптимизацией запросов никогда не приходится скучать. Она постоянно обеспечивает интересные задачи и возможность научиться новому. Я надеюсь, что эта глава дала вам не только советы о том, с чего начать, но и стимул практиковаться и развиваться дальше. В конце концов, настройка запросов — это самый эффективный способ повысить производительность системы.

Еще несколько советов: не создавайте отдельные индексы под каждый запрос. Проанализируйте рабочую нагрузку в системе и создайте индексы, которые пригодятся в нескольких запросах. Приступая к оптимизации, изучите наименее эффективные запросы и выявите распространенные шаблоны доступа.

Наконец, будьте осторожны с покрывающими индексами. Их не стоит делать слишком широкими. Опять же, в поиске по ключу нет ничего плохого, если он не оказывает существенного влияния на систему.

В следующей главе мы поговорим о высокой загрузке ЦП и о том, как ее снизить.

Резюме

SQL Server поддерживает три технологии хранения и обработки данных. Хранилище на основе строк — самый распространенный случай — хранит данные из всех столбцов вместе в несортированных кучах или отсортированных индексах на основе B-деревьев.

Индексы B-деревьев сортируют данные на основе ключей индекса. Кластеризованные индексы хранят данные из всех столбцов таблицы. Некластери-

зованные индексы хранят отдельную копию данных в отдельных физических индексах. Они ссылаются на кластеризованные индексы через идентификаторы строк, которые представляют собой значения ключа кластеризованного индекса. Когда данные отсутствуют в некластеризованном индексе, SQL Server просматривает B-дерево кластеризованного индекса, используя операцию поиска по ключу. При масштабировании эта операция становится очень ресурсоемкой.

SQL Server обращается к данным двумя способами. *Просмотр индекса* обычно считывает все строки из индекса. *Поиск по индексу* изолирует и обрабатывает подмножество строк, и он обычно более эффективен, чем просмотр. Составляйте запросы так, чтобы SQL Server мог использовать поиск по индексу. Стоит также анализировать эффективность поиска по индексу, убедившись, что при этом не обрабатываются слишком большие объемы данных.

SQL Server хранит информацию об индексах и распределении данных в статистике, с помощью которой он оценивает, сколько строк будет необходимо обработать каждому оператору в плане выполнения. Точная оценка количества элементов помогает SQL Server создавать эффективные планы выполнения. Актуальная статистика критически важна для правильной оценки количества элементов.

Анализируя планы выполнения, обратите внимание на эффективность операторов поиска по индексу, а также на выбор типа соединения и количество операций поиска по ключу и по идентификатору строки. Также проверьте оценку количества элементов: неправильная оценка — одна из самых распространенных причин низкого качества планов выполнения. Убедитесь, что статистика актуальна, добавьте необходимые индексы, а также упростите и переформулируйте запросы по необходимости.

Чек-лист устранения неполадок

- ☐ Проанализировать стратегию ведения статистики. Включить флаг трассировки `T2371`, чтобы уменьшить порог автоматического обновления статистики, если в системе есть базы данных с уровнем совместимости ниже 130 (SQL Server 2016).
- ☐ Проанализировать и улучшить стратегию обслуживания индексов.
- ☐ Убедиться, что статистика по отфильтрованным индексам обновляется достаточно часто. Решить, нужно ли обновлять ее еще чаще.
- ☐ Рассмотреть возможность перехода на последний уровень совместимости базы данных, особенно в SQL Server 2017 и более поздних версиях. В случае перехода обратить внимание на модель оценки количества элементов: возможно, с ней придется возиться отдельно.

- ☐ Проанализировать, как используются временные таблицы и табличные переменные (подробнее см. в главе 9).
- ☐ Выполнить рефакторинг предикатов, не поддерживающих поиск и аргументы (SARG), если они встречаются в критических запросах.
- ☐ Проверить правильность оценки количества элементов. Сократить использование функций из нескольких инструкций и других конструкций, которые могут повлиять на эту оценку.
- ☐ Проанализировать эффективность операторов поиска по индексу.
- ☐ Убедиться, что столбцы соединения содержат одинаковые типы данных, а тип соединения выбран правильно.
- ☐ По возможности создать покрывающие индексы, если в критических запросах оказалось много операций поисков по ключу.

ГЛАВА 6

Загрузка процессора

Мой первый проект по настройке SQL Server состоялся более 20 лет назад, и с тех пор я работал со многими системами. С годами я заметил интересную тенденцию. Раньше большинство систем, которые я оптимизировал, страдали в первую очередь из-за неполадок ввода/вывода. Конечно, были и другие проблемы, но обычно удавалось достичь удовлетворительных результатов, если снизить нагрузку на ввод/вывод путем настройки запросов и рефакторинга кода.

Несколько лет назад ситуация начала меняться. Хотя еще встречаются неоптимизированные запросы с интенсивным вводом/выводом, их влияние нивелируется высокопроизводительными флеш-накопителями с малой задержкой. Более того, доступность дешевого оборудования позволяет развертывать более крупные серверы, которые справляются с нагрузкой от большего количества пользователей. Зато в наше время часто возникает необходимость снизить нагрузку на ЦП.

В этой главе я расскажу о нескольких распространенных причинах повышенной нагрузки на ЦП и о том, как с ними бороться. Начнем с неоптимизированных запросов и неэффективного кода базы данных. Затем поговорим о накладных расходах на компиляцию запросов, а также о проблемах, связанных с кэшированием планов. В конце главы пойдет речь о преимуществах и недостатках параллелизма и о том, как настроить его параметры.

Неоптимизированные запросы и код T-SQL

Почему сервер потребляет много ресурсов процессора?

Причин может быть много, но я начну с самой очевидной и распространенной: неоптимизированные запросы. Неважно, насколько быстро работает дисковая подсистема. Неважно, достаточно ли у сервера памяти, чтобы кэшировать все данные в буферном пуле и не обращаться к физическому вводу/выводу. Неоптимизированные запросы все равно *увеличивают* нагрузку на ЦП.

Для сравнения: запрос, просматривающий 10 млн страниц данных, использует в миллион раз больше ресурсов ЦП, чем запрос, который просматривает лишь 10 страниц. Неважно, что каждое логическое чтение занимает всего несколько микросекунд процессорного времени: нагрузка стремительно нарастает, когда много пользователей запускают эти запросы параллельно.

Чтобы определить, какие запросы интенсивнее всего потребляют ресурсы ЦП, можно использовать методы, которые я обсуждал в главе 4, отсортировав данные по времени ЦП (исполнителя) и выбрав запросы, которые в первую очередь требуют оптимизации.

Однако не путайте *процессорное время* с *продолжительностью*. Хотя запросы с большим процессорным временем в среднем выполняются дольше, обратное неверно. Запрос может находиться под блокировкой и потреблять мало ресурсов ЦП, но при этом его выполнение займет много времени.

Пользователи довольны, когда запросы выполняются быстрее, но в моей практике продолжительность выполнения — не главный критерий выбора того, какие запросы оптимизировать. Если оптимизировать запросы с высоким потреблением ресурсов, то продолжительность их выполнения обычно тоже снижается.

Неэффективный код T-SQL

Неэффективный код T-SQL тоже нагружает процессор, потому что SQL Server интерпретирует код T-SQL, за исключением скомпилированных в собственном коде модулей In-Memory OLTP.

Не поймите меня неправильно: я не отговариваю вас от того, чтобы использовать хранимые процедуры и T-SQL. Преимущества хорошо спроектированных и реализованных модулей T-SQL значительно перевешивают нагрузку на ЦП. Но есть один случай, о котором стоит упомянуть особо: построчная обработка.

Независимо от того, как вы реализуете построчную обработку — с помощью курсоров или циклов, — она все равно неэффективна. Императивное построчное выполнение медленнее и требовательнее к ресурсам ЦП, чем декларативная логика на основе множеств. Бывают редкие ситуации, когда никак нельзя обойтись без построчной обработки, однако в остальных случаях ее стоит избегать.

Инструкции, вызывающие построчную обработку, не всегда кажутся самыми ресурсоемкими. Их можно обнаружить с помощью хранилища запросов, если оно включено. Также можно просмотреть статистику выполнения на основе кэша планов для модулей T-SQL с помощью представлений `sys.dm_exec_procedure_stats`, `sys.dm_exec_function_stats` и `sys.dm_exec_trigger_stats` (о них мы говорили в главе 4), чтобы найти модули, которые потребляют больше всего

ресурсов. Проанализируйте, как они работают, обращая особое внимание на построчную логику.

Другие конструкции T-SQL тоже влияют на потребление ЦП: например, поддержка JSON и особенно XML сильно загружает ЦП. Частично структурированные данные лучше анализировать на стороне клиента, а не в SQL Server. Также проще и дешевле масштабировать серверы приложений, потому что при этом не нужно платить за лицензию SQL Server.

Не упускайте из виду CLR, код внешних языков[1] и расширенные хранимые процедуры со сложной логикой. Избегайте обильных вызовов функций, особенно если это пользовательские функции T-SQL. Они добавляют накладные расходы и снижают эффективность планов выполнения, если не встраиваются.

Следите за тем, как используются представления. В зависимости от схемы и определения базы данных в них могут выполняться лишние соединения, ради которых происходят обращения к таблицам, ненужным для запроса. Особенно часто так бывает, если в таблицах не определены правильные внешние ключи.

Сценарии для контроля за загрузкой ЦП

Я покажу пару сценариев, которые помогут бороться с высокой загрузкой процессора. Первый сценарий, приведенный в листинге 6.1, показывает загрузку процессора на сервере за последние 256 минут. Данные измеряются раз в минуту, так что короткие всплески нагрузки на ЦП, возникающие между измерениями, могут оказаться неучтенными.

Листинг 6.1. Получение истории загрузки ЦП

```
DECLARE
  @now BIGINT;

SELECT @now = cpu_ticks / (cpu_ticks / ms_ticks)
FROM sys.dm_os_sys_info WITH (NOLOCK);

;WITH RingBufferData([timestamp], rec)
AS
(
  SELECT [timestamp], CONVERT(XML, record) AS rec
  FROM sys.dm_os_ring_buffers WITH (NOLOCK)
  WHERE
    ring_buffer_type = N'RING_BUFFER_SCHEDULER_MONITOR' AND
    record LIKE N'%<SystemHealth>%'
)
,Data(id, SystemIdle, SQLCPU, [timestamp])
```

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

```
AS
(
  SELECT
    rec.value('(./Record/@id)[1]', 'int')
    ,rec.value
    ('(./Record/SchedulerMonitorEvent/SystemHealth/SystemIdle)[1]','int')
    ,rec.value
    ('(./Record/SchedulerMonitorEvent/SystemHealth/ProcessUtilization)[1]','int')
    ,[timestamp]
  FROM RingBufferData
)
SELECT TOP 256
  dateadd(MS, -1 * (@now - [timestamp]), getdate()) AS [Event Time]
  ,SQLCPU AS [SQL Server CPU Utilization]
  ,SystemIdle AS [System Idle]
  ,100 - SystemIdle - SQLCPU AS [Other Processes CPU Utilization]
FROM Data
ORDER BY id desc
OPTION (RECOMPILE, MAXDOP 1);
```

На рис. 6.1 представлен результат работы кода. Данные в столбце `[Other Processes CPU Utilization]` показывают загрузку ЦП из-за других процессов в системе, помимо SQL Server. Если эта загрузка велика, разберитесь, какие процессы выполняются на сервере и создают ее.

	Event Time	SQL Server CPU Utilization	System Idle	Other Processes CPU Utilization
1	2021-03-27 09:44:52.903	61	35	4
2	2021-03-27 09:43:52.877	62	36	2
3	2021-03-27 09:42:52.863	56	42	2
4	2021-03-27 09:41:52.847	68	30	2
5	2021-03-27 09:40:52.833	77	20	3
6	2021-03-27 09:39:52.813	63	34	3
7	2021-03-27 09:38:52.800	64	33	3
8	2021-03-27 09:37:52.787	62	35	3

Рис. 6.1. Вывод сценария, получающего историю загрузки ЦП

Листинг 6.2 помогает анализировать загрузку процессора для каждой базы данных. Это может быть полезно, если на сервере размещено несколько баз и вы планируете рассредоточить высоконагруженные базы данных по разным серверам. (Отметим, что сценарий использует данные кэша планов, поэтому вывод может быть неточен.)

Листинг 6.2. Нагрузка на ЦП по отдельным базам данных

```
;WITH DBCPU
AS
(
```

```
    SELECT
        pa.DBID, DB_NAME(pa.DBID) AS [DB]
        ,SUM(qs.total_worker_time/1000) AS [CPUTime]
    FROM
        sys.dm_exec_query_stats qs WITH (NOLOCK)
        CROSS APPLY
        (
            SELECT CONVERT(INT, value) AS [DBID]
            FROM sys.dm_exec_plan_attributes(qs.plan_handle)
            WHERE attribute = N'dbid'
        ) AS pa
    GROUP BY pa.DBID
)
SELECT
    [DB]
    ,[CPUTime] AS [CPU Time (ms)]
    ,CONVERT(decimal(5,2), 1. * [CPUTime] /
        SUM([CPUTime]) OVER() * 100.0) AS [CPU Percent]
FROM DBCPU
WHERE DBID <> 32767 -- ResourceDB
ORDER BY [CPUTime] DESC;
```

На рис. 6.2 показан вывод с действующего сервера.

	DB	CPU Time (ms)	CPU Percent
1	Appdb	55332331	73.69
2	AppDW	16190787	21.56
3	master	2264420	3.02
4	DBA	918845	1.22

Рис. 6.2. Вывод сценария, измеряющего загрузку ЦП по отдельным базам данных

Шаблоны неоптимизированных запросов

В случае неоптимизированных запросов существуют два различных шаблона, которые могут вызвать высокую нагрузку на ЦП. Я называю их «злостные нарушители» и «смерть от тысячи порезов» (это не стандартные термины, а мои собственные обозначения для удобства).

Злостные нарушители

«Злостные нарушители» — это ситуация, когда один или несколько ресурсоемких, длительных запросов создает большую нагрузку на ЦП. Представьте себе неоптимизированные запросы с планами параллельного выполнения, которые просматривают миллионы строк и выполняют сортировку и агрегирование.

Они могут «положить» сервер, особенно когда несколько таких запросов выполняются одновременно.

К счастью, «злостных нарушителей» легко обнаружить в режиме реального времени, если запросить представление `sys.dm_exec_requests` и проанализировать столбец `cpu_time` (можно использовать код из листинга 2.3).

Внимание: код в листинге 2.3 отфильтровывает системные процессы с `session_id` ниже 50. В некоторых случаях имеет смысл удалить этот фильтр и просмотреть все сеансы, запущенные на сервере. Имейте в виду, что некоторые сеансы могут выполняться с момента запуска SQL Server и у них будет большое совокупное значение `cpu_time`, так что обращайте внимание на время начала запроса.

Смерть от тысячи порезов

В случае «смерти от тысячи порезов» нагрузка на сервер создается большим количеством одновременно выполняющихся запросов. Каждый запрос может быть относительно небольшим и даже оптимизированным, но их огромное число пожирает ресурсы ЦП и перенагружает сервер.

Эта ситуация сложнее предыдущей. Оптимизация запросов может помочь, однако если придется оптимизировать очень много запросов, это потребует значительного времени и усилий. Чтобы достичь результата, может понадобиться масштабный рефакторинг схем баз данных, кода и приложений.

Не забывайте, что конечная цель — снизить нагрузку на сервер. Давайте поговорим о других факторах этой нагрузки, начиная с компиляции запросов и процессов кэширования планов. Уменьшение накладных расходов, связанных с этими факторами, может облегчить общую нагрузку на сервер.

Компиляция запросов и кэширование планов

Каждый раз, когда вы создаете запрос к базе данных, SQL Server должен его скомпилировать и оптимизировать. Это ресурсоемкий процесс, поэтому SQL Server пытается минимизировать количество компиляций, кэшируя планы выполнения для последующего повторного использования. Помимо обычных клиентских запросов и пакетов, он кэширует планы различных объектов, таких как хранимые процедуры, триггеры и пользовательские функции. Область памяти, в которой они хранятся, называется *кэшем планов*.

SQL Server использует разные алгоритмы, чтобы определить, какие планы удалять из кэша, если не хватает памяти. Для нерегламентированных запросов

выбор зависит от того, как часто план используется повторно. Для других типов планов учитываются также затраты на создание плана.

SQL Server перекомпилирует запросы, когда подозревает, что текущие кэшированные планы устарели. Это может произойти, если план ссылается на объекты, чьи схемы изменились, или из-за устаревшей статистики. SQL Server проверяет, не устарела ли статистика, когда ищет план из кэша, и перекомпилирует запрос, если оказалось, что она устарела. Эта перекомпиляция, в свою очередь, запускает обновление статистики.

Кэширование планов и их повторное использование может значительно сократить количество компиляций и загрузку процессора, как я продемонстрирую далее в этой главе. Тем не менее из-за этого бывают и проблемы. Рассмотрим наиболее распространенные из них, начиная с *чувствительности планов к параметрам*. Она проявляется при так называемом *сканировании параметров* (*parameter sniffing*).

Планы, чувствительные к параметрам

За исключением простейших запросов, SQL Server может по-разному создавать планы выполнения для одного и того же запроса. Он может использовать разные индексы для доступа к данным, выбирать разные типы соединений, разные операторы и стратегии выполнения.

По умолчанию SQL Server анализирует (*сканирует*) значения параметров во время оптимизации, после чего генерирует и кэширует оптимальный план для этих значений. В таком поведении нет ничего плохого; однако если ваши данные распределены неравномерно, то в кэше может оказаться план, который оптимален для аномальных, редко используемых значений параметров, но крайне неэффективен для запросов с более типичными параметрами.

Уверен, что все вы сталкивались с ситуацией, когда некоторые запросы или хранимые процедуры внезапно начинают выполняться намного дольше, даже если в системе ничего не менялось. В большинстве случаев это происходит из-за перекомпиляции запросов после того, как статистика обновилась в результате сканирования параметров.

Рассмотрим пример. Сценарий в листинге 6.3 создает таблицу и заполняет в ней 1 млн строк, равномерно распределенных по 10 значениям `StoreId` (чуть более 100 000 строк на каждый `StoreId`), а также 10 строк со значением `StoreId`, равным 99.

Запускайте эту демонстрацию на базе данных с уровнем совместимости 150 (SQL Server 2019) или ниже. В SQL Server 2022 код может вести себя иначе; я расскажу об этом позже в этой главе.

Листинг 6.3. Планы, чувствительные к параметрам: создание таблицы

```
CREATE TABLE dbo.Orders
(
    OrderId INT NOT NULL IDENTITY(1,1),
    OrderNum VARCHAR(32) NOT NULL,
    CustomerId UNIQUEIDENTIFIER NOT NULL,
    Amount MONEY NOT NULL,
    StoreId INT NOT NULL,
    Fulfilled BIT NOT NULL
);

;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 строки
,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 строки
,N3(C) AS (SELECT 0 FROM N2 AS T1 CROSS JOIN N2 AS T2) -- 16 строк
,N4(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N3 AS T2 CROSS JOIN N2 AS T3)
,N5(C) AS (SELECT 0 FROM N4 AS T1 CROSS JOIN N4 AS T2 ) -- 1,048,576 строк
,IDs(ID) AS (SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) FROM N5)
INSERT INTO dbo.Orders(OrderNum, CustomerId, Amount, StoreId, Fulfilled)
    SELECT
        'Order: ' + CONVERT(VARCHAR(32),ID)
        ,NEWID()
        ,ID % 100
        ,ID % 10
        ,1
    FROM IDs;

INSERT INTO dbo.Orders(OrderNum, CustomerId, Amount, StoreId, Fulfilled)
    SELECT TOP 10 OrderNum, CustomerId, Amount, 99, 0
    FROM dbo.Orders
    ORDER BY OrderId;

CREATE UNIQUE CLUSTERED INDEX IDX_Orders_OrderId
ON dbo.Orders(OrderId);

CREATE NONCLUSTERED INDEX IDX_Orders_CustomerId
ON dbo.Orders(CustomerId);

CREATE NONCLUSTERED INDEX IDX_Orders_StoreId
ON dbo.Orders(StoreId);
```

Теперь создадим хранимую процедуру, которая вычисляет общую сумму продаж для определенного магазина (листинг 6.4). В этом примере я использую именно хранимую процедуру, однако параметризованные запросы, вызываемые из клиентских приложений, будут вести себя точно так же.

Листинг 6.4. Планы, чувствительные к параметрам: хранимая процедура

```
CREATE PROC dbo.GetTotalPerStore(@StoreId int)
AS
    SELECT SUM(Amount) as [Total Amount]
    FROM dbo.Orders
    WHERE StoreId = @StoreId;
```

Если при текущем распределении данных хранимая процедура вызывается с любым `@StoreId`, отличным от 99, то в оптимальный план выполнения входит просмотр кластеризованного индекса в таблице. Однако если `@StoreId=99`, то лучшим планом выполнения будет поиск по индексу `IDX_Orders_StoreId` с последующим поиском по ключу.

Вызовем хранимую процедуру дважды: первый раз со `@StoreId=5` и второй раз со `@StoreId=99`, как показано в листинге 6.5.

Листинг 6.5. Планы, чувствительные к параметрам: вызов процедуры (тест 1)

```
EXEC dbo.GetTotalPerStore @StoreId = 5;
EXEC dbo.GetTotalPerStore @StoreId = 99;
```

Как видно из плана выполнения на рис. 6.3, SQL Server компилирует хранимую процедуру, кэширует план при первом вызове и затем повторно использует план. Этот план менее эффективен для второго вызова со `@StoreId=99`, однако он вполне пригоден, когда такие вызовы редки: при таком распределении данных это весьма ожидаемо.

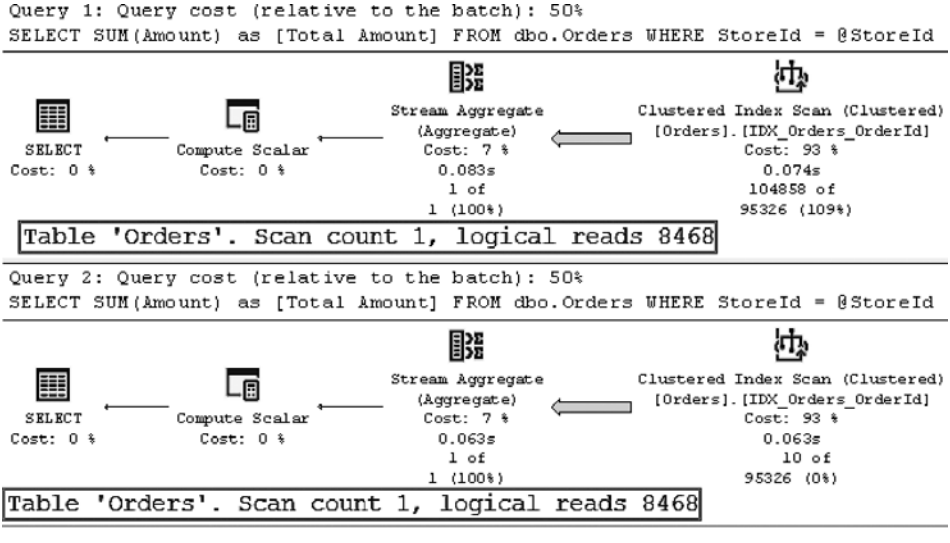


Рис. 6.3. Планы выполнения запросов (тест 1)

Теперь посмотрим, что произойдет, если поменять местами эти вызовы, когда план *не* кэшируется (листинг 6.6). Я очищу кэш плана с помощью команды `DBCC FREEPROCCACHE` (не запускайте эту демонстрацию на рабочем сервере!). Учтите, что то же самое может случиться, если повторная компиляция запроса произойдет из-за обновления статистики.

Листинг 6.6. Планы, чувствительные к параметрам: вызов процедуры (тест 2)

```
DBCC FREEPROCCACHE;
EXEC dbo.GetTotalPerStore @StoreId = 99;
EXEC dbo.GetTotalPerStore @StoreId = 5;
```

Как видно на рис. 6.4, теперь SQL Server кэширует план, скомпилированный для значения параметра `@StoreId=99`. Этот план эффективен, когда хранимая процедура вызывается именно с этим значением, однако крайне неэффективен при других значениях `@StoreId`.

Неэффективные планы, чувствительные к параметрам, часто становятся «злостными нарушителями», которые перенагружают ЦП. Как я уже упоминал, соответствующие запросы можно обнаружить с помощью представления `sys.dm_exec_requests` (листинг 2.3) и перекомпилировать, чтобы устранить проблему.

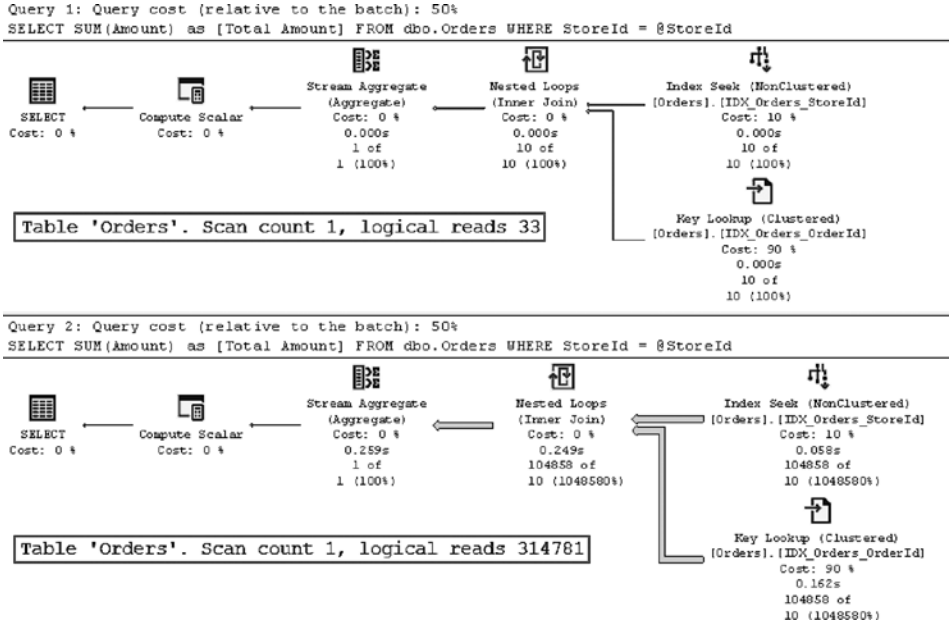


Рис. 6.4. Планы выполнения запросов (тест 2)

Хранимые процедуры и другие модули T-SQL можно принудительно перекомпилировать с помощью хранимой процедуры `sp_recompile`. Для нерегламентированных запросов можно вызвать `DBCC FREEPROCCACHE`, передав в качестве параметра `plan_handle` или `sql_handle`. Наконец, если у вас включено хранилище запросов, в нем можно явно задать более эффективный план выполнения.

Очевидно, лучше устранить первопричину проблемы. Во-первых, посмотрите, нет ли возможности скорректировать запросы так, чтобы избавиться от чувствительности планов к параметрам. Планы, чувствительные к параметрам, обычно получаются из-за того, что невозможно сгенерировать эффективные планы, не зависящие от значений параметров. Например, если включить столбец `Amount` в индекс `IDX_Orders_StoreId`, то этот индекс станет покрывающим. SQL Server сможет использовать его для всех значений параметров независимо от того, сколько строк будет прочитано, потому что операция *Key Lookup* больше не требуется.

В SQL Server 2017 или более поздних версиях можно воспользоваться *автоматической коррекцией плана*, которая входит в инструменты *автоматической настройки*. Когда эта функция включена, SQL Server может обнаруживать проблемы сканирования параметров и автоматически применять последний успешный план, который использовался до возникновения регресса.

Автоматическая коррекция плана опирается на функцию *Force Plan* в хранилище запросов, так что оно должно быть включено в базе данных. Более того, включить его нужно с помощью команды `ALTER DATABASE SET AUTOMATIC_TUNING (FORCE_LAST_GOOD_PLAN = ON)`. Подробнее об этом рассказывается в документации Microsoft[1].

В SQL Server 2022 появилась новая функция — *оптимизация планов, чувствительных к параметрам*. В базах данных с уровнем совместимости 160 и выше SQL Server может кэшировать несколько активных планов выполнения для параметризованных пакетов SQL и модулей T-SQL. SQL Server вычисляет значения параметров во время выполнения, выбирая оптимальный план для каждого выполнения.

Оптимизация планов, чувствительных к параметрам, добавляет небольшие накладные расходы во время компиляции запроса и увеличивает размер кэша планов. Поэтому SQL Server кэширует несколько планов, только когда обнаруживает неравномерное распределение данных, которое может привести к проблемам в планах, чувствительных к параметрам. Кэширование множественных планов зависит и от других факторов, однако, когда эта книга была подписана в печать, данная функция еще не была выпущена. Изучите ее внимательнее, если вы используете SQL Server 2022 или базу данных Azure SQL.

Если ни один из предыдущих вариантов не помогает, можно принудительно перекомпилировать хранимую процедуру с помощью команды `EXECUTE WITH RECOMPILE` или инструкцию с помощью предложений `OPTION (RECOMPILE)`. В листинге 6.7 показан последний подход.

[1] https://oreil.ly/GssX2

Листинг 6.7. Планы, чувствительные к параметрам: перекомпиляция на уровне инструкций

```
ALTER PROC dbo.GetTotalPerStore(@StoreId int)
AS
    SELECT SUM(Amount) as [Total Amount]
    FROM dbo.Orders
    WHERE StoreId = @StoreId
    OPTION (RECOMPILE);
GO
EXEC dbo.GetTotalPerStore @StoreId = 99;
EXEC dbo.GetTotalPerStore @StoreId = 5;
```

На рис. 6.5 видно, что SQL Server перекомпилирует запрос, сканирующий параметры, во время каждого вызова.

Принудительная перекомпиляция позволяет получить наиболее эффективные планы выполнения для каждого вызова ценой постоянных накладных расходов на перекомпиляцию. Такой подход оправдан для редко выполняемых запросов, однако при частом выполнении он может заметно увеличить нагрузку на ЦП, как я покажу позже в этой главе.

Эту проблему можно решить, применив другое указание: `OPTIMIZE FOR`. Оно определяет, какие значения параметров оптимизатор запросов будет использовать во время оптимизации. Например, с указанием `OPTIMIZE FOR (@StoreId=5)` оптимизатор запросов будет не сканировать `@StoreId`, а оптимизировать его для значения 5.

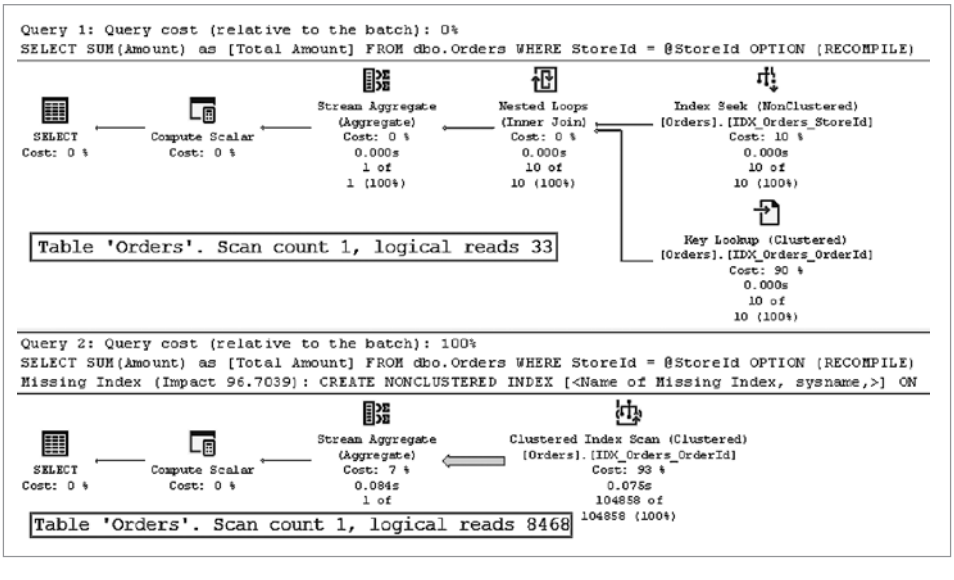


Рис. 6.5. Планы выполнения запросов (перекомпиляция на уровне инструкций)

Нетрудно догадаться, что указание `OPTIMIZE FOR` несет риск в случае, если распределение данных изменится. Например, если магазин с `@StoreId=5` прекратит существовать, то планы выполнения станут крайне неэффективными.

К счастью, у этого указания есть и другая форма: `OPTIMIZE FOR UNKNOWN`. В этом случае SQL Server выполняет оптимизацию на основе наиболее часто встречающегося значения в таблице. В нашем случае это указание привело бы к плану выполнения со сканированием кластеризованного индекса, что ожидаемо при таком распределении данных в таблице.

В листинге 6.8 приведен пример указания `OPTIMIZE FOR UNKNOWN`. Обратите внимание, что его можно использовать с подмножествами параметров запроса.

Листинг 6.8. Планы, чувствительные к параметрам: указание `OPTIMIZE FOR UNKNOWN`

```
ALTER PROC dbo.GetTotalPerStore(@StoreId int)
AS
    SELECT SUM(Amount) as [Total Amount]
    FROM dbo.Orders
    WHERE StoreId = @StoreId
    OPTION (OPTIMIZE FOR (@StoreId UNKNOWN));
GO
```

На уровне запроса можно использовать указания `OPTIMIZE FOR UNKNOWN` (во всех версиях SQL Server после 2008) или `DISABLE_PARAMETER_SNIFFING` (в SQL Server 2016 и более поздних версиях). Оба указания работают одинаково. В SQL Server 2016 также можно управлять сканированием параметров на уровне базы данных с помощью ее параметра `PARAMETER_SNIFFING`. Наконец, можно отключить сканирование параметров на уровне сервера флагом трассировки `T4136`. Он также работает в версиях SQL Server до 2016.

По моему опыту, отключение сканирования параметров приводит к лучшим и более стабильным планам выполнения в многопользовательских системах и в системах с очень неравномерным распределением данных. Ваш опыт может отличаться, но этот метод стоит попробовать, если у вас в системе много планов, чувствительных к параметрам.

Кэширование неэффективных планов, чувствительных к параметрам, увеличивает нагрузку на ЦП. К сожалению, это не единственная проблема, с которой можно столкнуться при кэшировании планов.

Независимость от значений параметров

Кэшированные планы выполнения должны быть пригодными для всех возможных комбинаций параметров в будущих вызовах. Поэтому если план предполагается кэшировать, то даже при сканировании параметров SQL Server не

сгенерирует план, который нельзя будет использовать с некоторыми значениями параметров.

Это звучит немного запутанно, поэтому позвольте мне продемонстрировать этот эффект на простом примере. В листинге 6.9 показан очень распространенный и обычно низкоэффективный шаблон: хранимая процедура принимает необязательные параметры, используя для них всех один и тот же запрос.

Листинг 6.9. Независимость от значений параметров

```
CREATE PROC dbo.SearchOrders
(
    @StoreId INT = NULL
    ,@CustomerId UNIQUEIDENTIFIER = NULL
)
AS
    SELECT OrderId, CustomerId, Amount, Fulfilled
    FROM dbo.Orders
    WHERE
        ((@StoreId IS NULL) OR (StoreId = @StoreId)) AND
        ((@CustomerId IS NULL) OR (CustomerId = @CustomerId));
GO

EXEC dbo.SearchOrders
    @StoreId = 99
    ,@CustomerId = 'A65C047D-5B08-4041-B2FE-8E3DD6570B8A';
```

Независимо от того, какие параметры использовались во время компиляции, получится план, аналогичный показанному на рис. 6.6. Несмотря на то что в обоих столбцах, `CustomerId` и `StoreId`, есть и индексы, и предикаты с поддержкой SARG, SQL Server выполняет *просмотр индекса* (*Index Scan*) вместо *поиска по индексу* (*Index Seek*). К сожалению, SQL Server не может выполнять поиск по индексу, потому что план должен годиться для кэширования и повторного использования в будущем. Этот план не годился бы, если бы не был указан *предикат поиска* (в данном случае параметр `@StoreId`).

Перекомпиляция на уровне инструкций с указанием `OPTION (RECOMPILE)` решает проблему — опять же, ценой дополнительных затрат на компиляцию. Как уже отмечалось, для редко выполняемых запросов эти затраты могут быть приемлемыми.

Как вариант, можно переписать код, включив туда инструкции `IF`, которые охватывают все возможные комбинации параметров. Тогда SQL Server будет кэшировать план для каждой инструкции. Это может сработать в простых ситуациях, однако код быстро выйдет из-под контроля по мере роста числа параметров.

Наконец, во многих случаях нет ничего плохого в том, чтобы применять динамический SQL. В листинге 6.10 показано, как это можно сделать. Естественно, не забывайте о безопасности и используйте параметры, чтобы предотвратить SQL-инъекции.

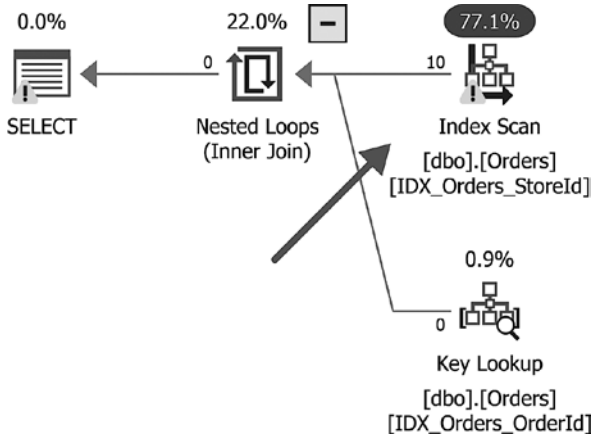


Рис. 6.6. План выполнения хранимой процедуры: независимость от значений параметров

Листинг 6.10. Реализация динамического SQL

```
ALTER PROC dbo.SearchOrders
(
    @StoreId INT = NULL
    ,@CustomerId UNIQUEIDENTIFIER = NULL
)
AS
BEGIN
    DECLARE
    @SQL nvarchar(max) =
N'SELECT OrderId, CustomerId, Amount, Fulfilled
FROM dbo.Orders
WHERE
    (1=1)' +
    IIF(@StoreId IS NOT NULL, N'AND (StoreId = @StoreId)','') +
    IIF(@CustomerId IS NOT NULL, N'AND (CustomerId = @CustomerId)','');

    EXEC sp_executesql
        @SQL = @SQL
        ,@Params = N'@StoreId INT, @CustomerId UNIQUEIDENTIFIER'
        ,@StoreId = @StoreId, @CustomerId = @CustomerId;
END
```

Существуют и другие ситуации, когда кэширование и повторное использование планов могут сделать их неэффективными. Один случай, который часто упускают из виду, связан с отфильтрованными индексами.

Запрос в листинге 6.11 не обращается к отфильтрованному индексу, даже если вызывать его с параметром `@Fulfilled = 0`. Это происходит потому, что если бы кэшированный план выполнения использовал отфильтрованный индекс, этот план не был бы действительным для `@Fulfilled = 1`.

Листинг 6.11. Запрос, который не обращается к отфильтрованному индексу

```
CREATE NONCLUSTERED INDEX IDX_Orders_ActiveOrders_Filtered
ON dbo.Orders(OrderId)
INCLUDE(Fulfilled)
WHERE Fulfilled = 0;
GO

DECLARE
    @Fulfilled bit = 0;

SELECT COUNT(*) AS [Active Order Count]
FROM dbo.Orders
WHERE Fulfilled = @Fulfilled;
```

Всегда добавляйте *все* столбцы из фильтра в ключевые или неключевые столбцы в отфильтрованных индексах. Это приводит к более эффективным планам выполнения.

К сожалению, эта проблема также может возникать из-за автоматической параметризации, особенно когда база данных использует принудительную параметризацию. Мы поговорим об этом подробнее в этой главе, но позже. А пока давайте посмотрим на компиляцию и накладные расходы, которые она создает.

Компиляция и параметризация

Как вы уже знаете, SQL Server кэширует и повторно использует планы выполнения для модулей T-SQL, а также для нерегламентированных клиентских запросов и пакетов. Однако в случае нерегламентированных запросов планы повторно используются только в случае *идентичных* запросов. Это обусловлено несколькими факторами.

Прежде всего идентичные запросы должны быть *абсолютно* одинаковыми, то есть соблюдать полное посимвольное соответствие. Посмотрите на запросы в листинге 6.12. Только два из них идентичны (первый и второй), хотя логически все четыре запроса одинаковы.

Листинг 6.12. Идентичные запросы

```
SELECT COUNT(*) FROM dbo.Orders WHERE StoreId = 99;
SELECT COUNT(*) FROM dbo.Orders WHERE StoreId = 99;
SELECT COUNT(*) FROM dbo.Orders WHERE StoreId=99;
select count(*) from dbo.Orders where StoreId = 99;
```

Кроме того, на повторное использование плана влияют некоторые параметры SET, в том числе ANSI_NULL_DLFT_OFF, ANSI_NULL_DLFT_ON, ANSI_NULLS, ANSI_PADDING, ANSI_WARNING, ARITHABORT, CONCAT_NULL_YELDS_NULL, DATEFIRST, DATEFORMAT,

FORCEPLAN, LANGUAGE, NO_BROWSETABLE, NUMERIC_ROUNDABORT, а также QUOTED_IDENTIFIER. Планы, сгенерированные с одним набором параметров SET, не будут повторно использоваться в сеансах, где действует другой набор.

Возможно, вы заметили, что я постоянно подчеркиваю тот факт, что компиляция и оптимизация запросов — ресурсоемкие процессы, и они могут интенсивно использовать ЦП при большой нерегламентированной рабочей нагрузке. Чтобы продемонстрировать это, я создал небольшое приложение, которое запускает простые запросы из листинга 6.13 в цикле в несколько потоков. Приложение можно скачать в составе сопутствующих материалов книги.

В первом тесте приложение запускает нерегламентированные запросы, используя непараметризованные значения CustomerId (запросы формируются в приложении). Каждый запрос в вызове уникален и должен быть скомпилирован. Во втором тесте фигурирует параметризованный запрос. План для этого запроса можно повторно использовать в разных вызовах.

Листинг 6.13. Нерегламентированная и параметризованная рабочая нагрузка

```
-- Тест 1
SELECT TOP 1 OrderId
FROM dbo.Orders
WHERE CustomerId = '<ID Generated in the app>';
  -- Идентификатор, сгенерированный в приложении

-- Тест 2
SELECT TOP 1 OrderId
FROM dbo.Orders
WHERE CustomerId = @CustomerId;
```

Оба запроса предельно просты. Кроме того, я запускал их в тестовой среде, где объем памяти позволял кэшировать всю таблицу и не использовать физический ввод/вывод.

На рис. 6.7 показаны метрики производительности, собранные во время тестов. Как видите, во втором тесте (справа) система смогла обработать почти в пять раз больше запросов в секунду, чем в первом.

Очевидно, этот сценарий — абсолютно искусственный, и в реальной жизни вы вряд ли встретите ситуацию, когда SQL Server вынужден тратить бóльшую часть времени на компиляцию запросов. Тем не менее в системах с большой нерегламентированной рабочей нагрузкой компиляция может существенно влиять на загрузку ЦП. Она влияет еще и на память, о чем я расскажу в главе 7.

Существуют три счетчика производительности SQL Server:SQL Statistics, которые показывают пропускную способность системы и количество компиляций:

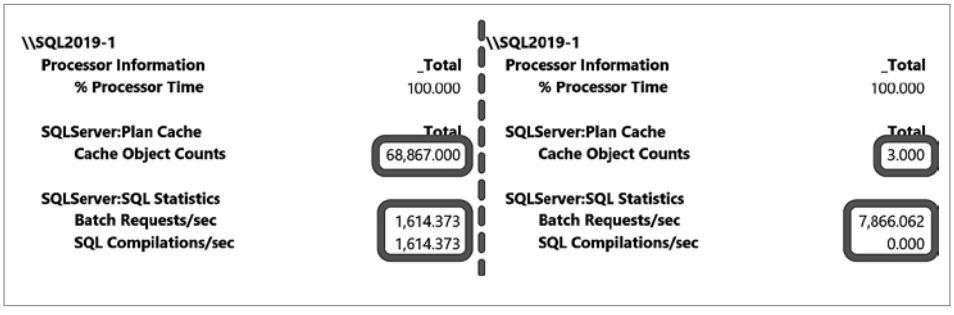


Рис. 6.7. Нерегламентированная и параметризованная рабочая нагрузка

`Batch Requests/sec`

Количество пакетов в секунду, которые получает SQL Server. Чем больше значение, тем больше нагрузка на систему и ее пропускная способность.

`SQL Compilations/sec`

Количество компиляций в секунду, которые выполняет SQL Server. Чем выше это число, тем больше компиляций и, следовательно, больше накладных расходов.

`SQL Re-Compilations/sec`

Количество перекомпиляций в секунду для ранее кэшированных планов выполнения. Перекомпиляции могут происходить из-за частых изменений исходных данных как в пользовательских, так и во временных таблицах.

В правильно настроенной OLTP-системе количество компиляций и перекомпиляций должно составлять небольшую долю от общего числа пакетных запросов. Если это не так, проанализируйте и сократите количество компиляций. (В главе 7 я расскажу, как анализировать данные кэша планов.)

Непараметризованные нерегламентированные клиентские запросы — наиболее частая причина компиляции. Как вы наверняка догадались, лучше всего переделать такие запросы и параметризовать их. К сожалению, для этого обычно требуется изменить код на стороне клиента, что не всегда возможно. К счастью, есть еще один вариант: *автоматическая параметризация*.

Автоматическая параметризация

SQL Server пытается уменьшить накладные расходы на компиляцию следующим образом: он заменяет константы в нерегламентированных запросах параметрами и кэширует скомпилированные планы, как если бы запросы были параметризованы. В результате кэшированные планы могут повторно

использоваться для похожих нерегламентированных запросов, содержащих разные константы.

Давайте рассмотрим пример и выполним запросы из листинга 6.14. Как и раньше, я очищаю кэш планов с помощью команды `DBCC FREEPROCCACHE`, чтобы уменьшить объем вывода.

Листинг 6.14. Автоматическая параметризация

```
DBCC FREEPROCCACHE
GO
SELECT * FROM dbo.Orders WHERE OrderId = 1;
GO
SELECT * FROM dbo.Orders WHERE OrderId = 2;
GO

SELECT
    p.usecounts, p.cacheobjtype, p.objtype, p.size_in_bytes, t.[text]
FROM
    sys.dm_exec_cached_plans p CROSS APPLY
        sys.dm_exec_sql_text(p.plan_handle) t
WHERE
    p.cacheobjtype LIKE 'Compiled Plan%' AND
    t.[text] LIKE '%Orders%'
ORDER BY
    p.objtype DESC
OPTION (RECOMPILE);
```

На рис. 6.8 показан вывод последней инструкции из кода. Как видите, в кэше планов оказались три записи: скомпилированный план, используемый для обоих автоматически параметризованных нерегламентированных запросов, а также два других объекта, называемых *запросами-оболочками*. Каждая оболочка занимает около 16 Кбайт памяти; она содержит информацию об исходном нерегламентированном запросе и ссылку на скомпилированный план.

	usecounts	cacheobjtype	objtype	size_in_bytes	text
1	2	Compiled Plan	Prepared	40960	(@1 tinyint)SELECT * FROM [dbo].[Orders] WHERE [OrderId]=@1
2	1	Compiled Plan	Adhoc	16384	SELECT * FROM dbo.Orders WHERE OrderId = 2;
3	1	Compiled Plan	Adhoc	16384	SELECT * FROM dbo.Orders WHERE OrderId = 1;

Рис. 6.8. Кэш планов после автоматической параметризации

Простая параметризация

По умолчанию SQL Server использует простую параметризацию (`SIMPLE`), которая очень консервативна. Она происходит, только если кэшированный план считается *безопасным для параметризации*. В эту категорию входят планы,

в которых форма плана и оценка количества элементов не изменятся, даже если поменяются значения констант или параметров.

Например, план с поиском по некластеризованному индексу и поиском по ключу по уникальному индексу безопасен, потому что поиск по некластеризованному индексу не вернет более одной строки независимо от значения параметра. Однако такая же операция с неуникальным индексом не считается безопасной. Различные значения параметров приведут к разным оценкам количества элементов, так что для некоторых значений параметров больше подойдет сканирование кластеризованного индекса.

Более того, существует много языковых конструкций, которые препятствуют автоматической параметризации: в их числе `IN`, `TOP`, `DISTINCT`, `JOIN`, `UNION` и подзапросы. На практике это означает, что большинство запросов не будут автоматически параметризоваться.

Принудительная параметризация

Помимо простой параметризации, SQL Server может использовать принудительную (`FORCED`). Ее можно включить на уровне базы данных с помощью команды `ALTER DATABASE SET PARAMETRIZATION FORCED` или на уровне запроса с помощью указания `PARAMETRIZATION FORCED`. В этом режиме SQL Server автоматически параметризует большинство нерегламентированных запросов (за очень немногими исключениями).

На рис. 6.9 показаны результаты первого теста (непараметризованные нерегламентированные запросы) из листинга 6.12 после того, как в базе данных была включена принудительная параметризация. Хотя пропускная способность системы по-прежнему значительно ниже, чем при грамотно параметризованной

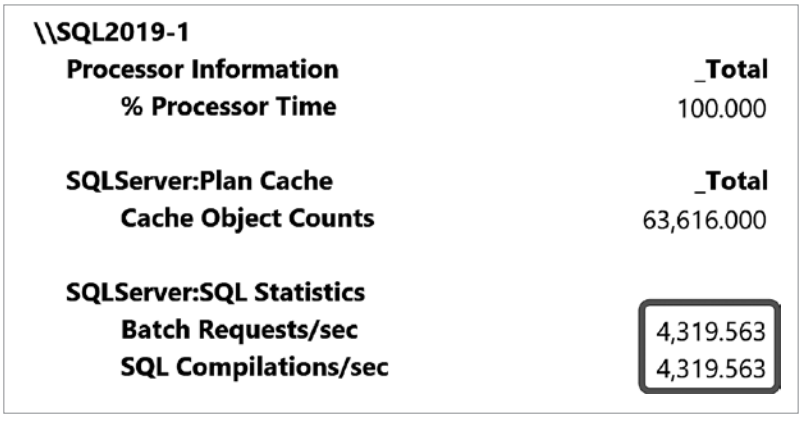


Рис. 6.9. Пропускная способность в случае принудительной параметризации

рабочей нагрузке, результаты все равно лучше, чем при простой параметризации (см. рис. 6.7), когда запросы не параметризовались автоматически. Хотя SQL Server и тратит какое-то процессорное время на автоматическую параметризацию, при этом он может повторно использовать кэшированный план выполнения и ему не нужно оптимизировать все запросы.

Принудительная параметризация может значительно снизить накладные расходы на компиляцию и загрузку ЦП в системах с большой нерегламентированной рабочей нагрузкой. Конечно, в разных системах будут разные результаты, но у меня бывали случаи, когда принудительная параметризация снижала нагрузку на ЦП на целых 25–30 %.

Но у принудительной параметризации есть и свои минусы. Когда она включена, SQL Server автоматически параметризует большинство нерегламентированных запросов, а от этого могут возникнуть планы, чувствительные к параметрам, и проблемы, связанные со сканированием параметров. Из-за этого некоторые нерегламентированные запросы регрессируют.

Рекомендую отключить сканирование параметров, если включена принудительная параметризация. Этот вариант не обязательно подойдет для *любой* системы, но в большинстве случаев полезен.

К счастью, не обязательно действовать по принципу «все или ничего». Как я уже упоминал, принудительную параметризацию можно включить на уровне запроса с помощью указания (hint) `PARAMETERIZATION FORCED`. Это полезно, если в системе всего несколько непараметризованных нерегламентированных запросов и вы не хотите применять принудительную параметризацию глобально.

Если нет доступа к исходному коду, это указание можно внедрить через структуры плана. В листинге 6.15 показано, как это сделать. В коде используются две хранимые процедуры. Первая, `sp_get_query_template`[1], создает шаблон запроса на основе образца запроса, указанного в качестве параметра. Создавая шаблон, в нерегламентированном запросе можно использовать любые константы. Вторая процедура, `sp_create_plan_guide`[2], создает структуру плана.

Листинг 6.15. Применение принудительной параметризации с помощью структуры плана

```
DECLARE
    @stmt NVARCHAR(MAX)
    ,@params NVARCHAR(MAX)
    ,@query NVARCHAR(MAX) =
N'SELECT TOP 1 OrderId FROM dbo.Orders
WHERE CustomerId = ''B970D68B-F88E-438B-9B04-6EDE47CC1D9A''';
```

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

[2] https://oreil.ly/cEJ4v

```
EXEC sp_get_query_template
    @querytext = @query
    ,@templatetext = @stmt output
    ,@params = @params output;

EXEC sp_create_plan_guide
    @type = N'TEMPLATE'
    ,@name = N'forced_parameterization_plan_guide'
    ,@stmt = @stmt
    ,@module_or_batch = null
    ,@params = @params
    ,@hints = N'OPTION (PARAMETERIZATION FORCED) ,;
```

Можно загрузить тестовое приложение из сопутствующих материалов этой книги и воспроизвести нагрузочные тесты, чтобы убедиться, что структура плана работает в базе данных, где используется простая параметризация.

В некоторых случаях нужно сделать наоборот: обеспечить простую параметризацию для отдельных запросов, использующих принудительную. Это может понадобиться, если у некоторых нерегламентированных запросов оказались планы, чувствительные к параметрам. В листинге 6.15 показано, как активировать простую параметризацию с помощью структуры плана, позволив нерегламентированному запросу использовать отфильтрованный индекс. В процедуру `sp_create_plan_guide` нужно передать инструкцию так, как если бы запрос уже был автоматически параметризован. Как показано в примере, инструкцию вместе с ее параметрами можно получить из кэша планов.

В листинге 6.16 я применяю структуру плана к первому запросу.

Листинг 6.16. Применение простой параметризации с помощью структуры плана

```
SELECT OrderId
FROM dbo.Orders
WHERE Fulfilled = 0;
GO

SELECT
    SUBSTRING(qt.text, (qs.statement_start_offset/2)+1,
    ((
        CASE qs.statement_end_offset
            WHEN -1 THEN DATALENGTH(qt.text)
            ELSE qs.statement_end_offset
        END - qs.statement_start_offset)/2)+1) AS SQL
    ,qt.text AS [Full SQL]
FROM
    sys.dm_exec_query_stats qs with (nolock)
        CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
WHERE
    qt.text like '%Fulfilled%'
OPTION(RECOMPILE, MAXDOP 1);

DECLARE
```

```
    @stmt NVARCHAR(MAX) =
        N'select OrderId from dbo . Orders where Fulfilled = @0'
    ,@params NVARCHAR(MAX) = N'@0 int'

-- Создаем структуру плана
EXEC sp_create_plan_guide
    @type = N'TEMPLATE'
    ,@name = N'simple_parameterization_plan_guide'
    ,@stmt = @stmt
    ,@module_or_batch = null
    ,@params = @params
    ,@hints = N'OPTION (PARAMETERIZATION SIMPLE)';
```

SQL Server не выполняет автоматическую параметризацию запросов в хранимых процедурах и других модулях T-SQL. Некоторые нерегламентированные запросы можно перенести в хранимые процедуры и таким образом избежать проблем с чувствительностью к параметрам, когда включена принудительная параметризация.

Наконец, повторюсь еще раз: *перекомпиляция может привести к значительным накладным расходам ЦП в системах с большими нерегламентированными рабочими нагрузками*. Обращайте на это внимание!

Параллелизм

SQL Server использует параллелизм, чтобы ускорить выполнение сложных запросов, разделяя запросы между несколькими процессорами (исполнителями). Это улучшает пользовательский опыт, потому что запросы выполняются быстрее. Но за все приходится платить, и параллелизм тоже всегда сопряжен с накладными расходами. Когда SQL Server создает планы параллельного выполнения, он вынужден делать дополнительную работу: разделять и объединять данные между несколькими рабочими процессами и координировать их выполнение.

Предположим, что запрос завершается за 1 секунду и план последовательного выполнения требует 1000 мс времени исполнителя. Тот же запрос с параллельным планом для четырех ЦП может быть выполнен за 300 мс, что в сумме потребует 1050 мс времени исполнителей. Управление параллелизмом требует дополнительной работы, и совокупное время ЦП всегда будет выше, чем при последовательном плане.

Эта дополнительная работа может повлиять на пропускную способность в высоконагруженных системах OLTP. Ускоренное выполнение одного запроса не даст преимуществ, если в очереди к ЦП еще много других запросов. Из-за накладных расходов на параллелизм запросам приходится дольше ждать, пока процессор освободится. Параллелизм хорош для формирования сложных отчетов и аналитических рабочих нагрузок, но в системах OLTP он может выйти боком, особенно когда процессоры сервера сильно загружены.

К сожалению, крайне сложно найти систему, в которой не смешивалась бы нагрузка обоих видов. Даже если вы реализуете выделенное хранилище данных и хранилище операционных данных (ODS, operational data storage), в исходных системах OLTP все равно будут выполняться некоторые отчеты и сложные запросы. В идеале эти рабочие нагрузки стоило бы разделить и запускать с разными настройками параллелизма.

Что еще хуже, конфигурация параллелизма в SQL Server по умолчанию далека от оптимальной. В планах параллельного выполнения можно использовать либо все ЦП (SQL Server 2017 и более ранние версии), либо до 16 ЦП (SQL Server 2019 и более поздние версии), причем эти планы создаются, когда стоимость запросов больше или равна 5 секундам. Выражаясь техническим языком, по умолчанию применяется *порог стоимости параллелизма*, или CTFP (cost threshold for parallelism), равный 5. Значение стоимости условно: оно не отражает содержательной информации и используется в качестве базовой метрики при оптимизации запросов. Как бы то ни было, в наше время 5 — крайне низкое значение, если учитывать, как растут объемы данных. Такой CFTP разрешает параллельные планы выполнения для огромного количества запросов.

Параллелизм проявляется в ожиданиях `CXPACKET`, `CXCONSUMER` и `EXCHANGE`. Но важно помнить, что параллелизм — не причина, а *симптом* проблемы. Высокая доля ожиданий, связанных с параллелизмом, лишь означает, что в системе много ресурсоемких запросов, что может быть совершенно нормально для данной рабочей нагрузки. Вместе с тем в системах OLTP такая цифра обычно говорит о том, что запросы плохо оптимизированы (стоимость оптимизированных запросов была бы меньше).

Чтобы увидеть стоимость отдельной инструкции, изучите свойство корневой инструкции в плане выполнения.

Обнаружив в системах OLTP много ожиданий, связанных с параллелизмом, я корректирую параметры параллелизма, после чего продолжаю устранять неполадки и настраивать запросы. Оптимизированные запросы имеют меньшую стоимость и, следовательно, вызывают меньше параллелизма. В некоторых случаях я даже отфильтровываю ожидания, связанные с параллелизмом, из статистики, чтобы получить более подробную картину прочих ожиданий.

Существует несколько подходов к настройке параметров параллелизма. В OLTP-системах я начинаю с того, чтобы установить `MAXDOP` в одну четвертую от количества доступных процессоров. Если у сервера много ЦП или он обрабатывает много запросов OLTP, я могу уменьшить этот параметр до одной восьмой или еще ниже. Наоборот, в хранилищах данных я иногда использую до половины доступных процессоров.

Еще важнее то, что я повышаю CTFP. Первым делом я часто задаю CTFP равным 50, но имеет смысл изучить стоимость запросов, чтобы подобрать еще более подходящее пороговое значение. В сопутствующих материалах книги можно найти слегка измененную версию листинга 4.1, которая предоставляет соответствующую информацию.

Изменив настройки, я отслеживаю загрузку ЦП, процент ожиданий сигналов и ожиданий, связанных с параллелизмом, а затем снова корректирую настройки. Одна из задач при этом — подобрать правильное значение CFTP, которое позволит SQL Server разделить разнородные рабочие нагрузки и уменьшить или даже предотвратить параллелизм в запросах OLTP.

Есть и другие, более тонкие механизмы управления параллелизмом. Например, можно отделить рабочие нагрузки OLTP от отчетов с помощью регулятора ресурсов (Resource Governor) и задать разные значения `MAXDOP` для разных групп нагрузок. Также в системах OLTP можно установить `MAXDOP=1`, а для отчетов добиваться параллелизма с помощью указания `MAXDOP`. Оба варианта потребуют от вас постоянного мониторинга системы и тесного сотрудничества с командами разработчиков.

В базах данных с уровнем совместимости 160 (SQL Server 2022) и выше появилась еще одна функция из семейства интеллектуальной обработки запросов (IOP): *обратная связь по степени параллелизма (DOP feedback)*. SQL Server отслеживает выполнение запросов с параллельными планами и может уменьшить степень параллелизма для отдельных запросов. Эта функция работает так же, как если бы вы применили указание запроса `MAXDOP`, но использует другой внутренний механизм управления параллелизмом.

Обратная связь по DOP работает подобно обратной связи по оценке количества элементов, которая рассматривалась в главе 5. SQL Server оценивает производительность запроса с новым значением DOP и в зависимости от результатов либо отбрасывает, либо сохраняет его. Он также может заново пересчитать DOP, уменьшив его вплоть до `MAXDOP = 2`. Впрочем, этот механизм не превратит план из параллельного в последовательный.

В любом случае, *не надо* устанавливать `MAXDOP` равным 1 для всех рабочих нагрузок системы. При этом вы просто замаскируете проблему. Помните, что параллелизм — это нормально; нужно лишь убедиться, что он применяется правильно.

Резюме

Проблемы высокой загрузки ЦП распространены в наши дни, потому что быстрые дисковые подсистемы и большие объемы памяти скрывают вред, который причиняют неоптимизированные запросы. Снижение нагрузки на ЦП часто становится основной целью настройки производительности.

Неоптимизированные запросы — главный фактор чрезмерной загрузки ЦП. Чем больше данных SQL Server приходится просматривать, тем больше ресурсов процессора он потребляет. Общая оптимизация запросов уменьшает эту проблему.

В системах с большими нерегламентированными рабочими нагрузками компиляция запросов может привести к значительной нагрузке на ЦП. Лучший способ бороться с этим — параметризовать запросы в коде. Также можно включить принудительную параметризацию на уровне отдельных запросов или базы данных.

К сожалению, параметризация может вызвать проблемы с планами, чувствительными к параметрам, когда SQL Server компилирует и кэширует планы для нетипичных значений параметров. При других комбинациях параметров такие планы могут оказаться крайне неэффективными. Во многих случаях удается улучшить ситуацию, если отключить сканирование параметров.

Обращайте внимание на степень параллелизма в вашей системе. Параллелизм совершенно нормален для отчетов и аналитических рабочих нагрузок, но нежелателен в системах OLTP, потому что управление параллелизмом добавляет накладные расходы.

Помните, что параллелизм — это признак ресурсоемких запросов. Их нужно оптимизировать, а не отключать параллелизм и тем самым скрывать проблему. Параметры параллелизма, используемые в SQL Server по умолчанию, неоптимальны и нуждаются в настройке.

В следующей главе вы узнаете, как в SQL Server устранять неполадки, связанные с памятью.

Чек-лист устранения неполадок

- ☐ Проанализировать и снизить нагрузку на ЦП от процессов, не связанных с SQL Server.
- ☐ Найти и оптимизировать «злостных нарушителей» — запросы, которые тратят больше всего времени рабочих процессов.
- ☐ Найти и оптимизировать наиболее ресурсоемкие хранимые процедуры и модули T-SQL.
- ☐ Оценить влияние компиляций. При перекрестной проверке данных могут пригодиться метрики кэша планов, которые мы обсудим в следующей главе.
- ☐ Параметризовать критические запросы. В наиболее сложных случаях тяжелых непараметризованных нерегламентированных рабочих нагрузок подумать о том, чтобы включить принудительную параметризацию и, возможно, отключить сканирование параметров.
- ☐ Настроить параметры параллелизма.

ГЛАВА 7

Проблемы с оперативной памятью

SQL Server может потреблять сотни гигабайт или даже терабайты оперативной памяти. Это совершенно нормально и даже хорошо, потому что при таких больших объемах памяти снижается потребность в физическом вводе/выводе и повторных компиляциях и производительность сервера повышается.

Эта глава — о работе с памятью в SQL Server. Для начала я расскажу, как SQL Server использует память, и дам несколько советов по ее конфигурации. Затем вы узнаете о процессе выделения памяти и научитесь анализировать, как ее потребляют внутренние компоненты SQL Server. Далее пойдет речь о предоставлении памяти для запросов и о том, как устранять неполадки, связанные с чрезмерным потреблением памяти. В завершение главы мы вкратце рассмотрим OLTP в памяти, затронув управление памятью в этой технологии и возможные проблемы с ее реализацией.

Использование и конфигурация памяти в SQL Server

SQL Server интенсивно использует память. По умолчанию он пытается выделять память по максимуму — чтобы каждой выполняемой операции досталось столько памяти, сколько ей нужно. Не вся память выделяется во время запуска; SQL Server также выделяет ее по мере необходимости: например, когда считывает страницы данных в буферный пул или сохраняет скомпилированные планы в кэше.

Часто бывает так, что SQL Server потребляет бóльшую часть памяти ОС. Это абсолютно нормальное явление. При правильной конфигурации SQL Server реагирует на запросы ОС и освобождает часть памяти процесса, когда нужно. Это состояние называется *нехваткой внешней памяти (external memory pressure)* и обычно возникает, когда у ОС недостаточно памяти для других приложений. В небольших количествах нехватка внешней памяти не обязательно опасна, но если SQL Server высвобождает чересчур много памяти, это может существенно повлиять на его производительность.

Некоторые события такого рода можно обнаружить, если настроить оповещение об ошибке 17890. Эта ошибка указывает на самый серьезный тип нехватки внешней памяти, когда часть памяти SQL Server выгружается на диск. В журнале ошибок появляется сообщение о том, сколько памяти выгружено: *A significant part of SQL Server process memory has been paged out* (*Выгружена значительная часть памяти процесса SQL Server*). Позже в этой главе я расскажу, как этого избежать.

Другая проблема — *нехватка внутренней памяти (internal memory pressure)* — может возникнуть, когда некоторые компоненты SQL Server потребляют много памяти, влияя на другие компоненты. В большинстве случаев SQL Server сам корректно обрабатывает такие ситуации, динамически регулируя использование внутренней памяти, но иногда возникают проблемы. Далее в этой главе вы узнаете, как с ними справляться.

Существует несколько способов отслеживать использование памяти на сервере. Два соответствующих счетчика производительности доступны в объекте *Memory Manager*; напоминаю, что их можно просмотреть с помощью утилиты `PerfMon` или представления `sys.dm_os_performance_counters`:

`Target Server Memory (KB)`

Целевой (идеальный) объем памяти, который должен потреблять SQL Server. Эта величина зависит от конфигурации, общего объема памяти, доступной для ОС, и некоторых других факторов.

`Total Server Memory (KB)`

Общий (текущий) объем памяти, который использует SQL Server.

В нормальных условиях значения этих счетчиков должны быть очень близкими. Но есть три случая, когда `Total Server Memory` может стать значительно ниже, чем `Target Server Memory`:

- Когда сервер разгоняется сразу после запуска. Это нормальное поведение.
- Когда аппаратной памяти очень много, и SQL Server не требуется вся доступная память. Это тоже нормально, но может свидетельствовать о неэффективном планировании мощностей.
- Во время нехватки памяти, когда SQL Server выгружает данные на диск. В этом случае нужно устранить неполадки.

Метрики памяти также можно получить из представлений `sys.dm_os_sys_memory`[1] и `sys.dm_os_process_memory`[2], которые содержат информацию о том, как используют память ОС и SQL Server соответственно. В листинге 7.1 приведен код, где применяются эти метрики.

[1] https://oreil.ly/xFS3u

[2] https://oreil.ly/1IBX4

Листинг 7.1. Анализ использования памяти ОС и SQL Server

```
SELECT
    total_physical_memory_kb / 1024 AS [Physical Memory (MB)]
    ,available_physical_memory_kb / 1024 AS [Available Memory (MB)]
    ,total_page_file_kb / 1024 AS [Page File Commit Limit (MB)]
    ,available_page_file_kb / 1024 AS [Available Page File (MB)]
    ,(total_page_file_kb – total_physical_memory_kb) / 1024
        AS [Physical Page File Size (MB)]
    ,system_cache_kb / 1024 AS [System Cache (MB)]
    /* Значения: LOW/HIGH/STEADY */
    ,system_memory_state_desc AS [System Memory State]
FROM sys.dm_os_sys_memory WITH (NOLOCK);

SELECT
    physical_memory_in_use_kb / 1024
        AS [SQL Server Memory Usage (MB)]
    ,locked_page_allocations_kb / 1024
        AS [SQL Server Locked Pages Allocation (MB)]
    ,large_page_allocations_kb / 1024
        AS [SQL Server Large Pages Allocation (MB)]
    ,memory_utilization_percentage
    ,available_commit_limit_kb
    ,process_physical_memory_low /* Может указывать на нехватку памяти */
    ,process_virtual_memory_low
FROM sys.dm_os_process_memory WITH (NOLOCK);
```

Историческую информацию о целевой и общей памяти можно получить из сеанса расширенного события `system_health`. Также она доступна в событии целевого объекта `sp_server_diagnostics_component_result` (его вывод частично показан в листинге 7.2). Эта информация может пригодиться для борьбы с необъяснимыми проблемами производительности, когда нужно проверить, испытывал ли сервер нехватку памяти в момент возникновения проблемы.

Эти данные также фиксируются скрытым сеансом расширенных событий и сохраняются в файлах XEL в папке журнала SQL Server. Имена файлов состоят из имен сервера и экземпляра, за которыми следует строка `SQLDIAG`.

Листинг 7.2. Событие `sp_server_diagnostics_component_result` в сеансе `system_health` (частичный вывод)

```
<resource lastNotification="RESOURCE_MEM_STEADY" outOfMemoryExceptions="0"
isAnyPoolOutOfMemory="0" processOutOfMemoryPeriod="0">
  <memoryReport name="Process/System Counts" unit="Value">
    <entry description="Available Physical Memory" value="65669554176" />
    <entry description="Available Virtual Memory" value="13879244778291" />
    <entry description="Available Paging File" value="67695706112" />
    <..>
  </memoryReport>
  <memoryReport name="Memory Manager" unit="KB">
    <entry description="Locked Pages Allocated" value="641593188" />
```

```
    <entry description="Large Pages Allocated" value="3248128" />
    <entry description="Target Committed" value="653261832" />
    <entry description="Current Committed" value="653263320" />
    <..>
  </memoryReport>
</resource>
```

Рассмотрим несколько приемов, которые помогают конфигурировать память в SQL Server.

Настройка памяти SQL Server

Существуют два хорошо известных параметра конфигурации, которые управляют использованием памяти в SQL Server:

Maximum Server Memory

Максимальный объем памяти, который может выделить SQL Server. Бывает, что SQL Server выделяет память сверх этого объема, но после этого он обнаруживает превышение и высвобождает лишнюю память.

Minimum Server Memory

Минимальный объем памяти, зарезервированный для экземпляра SQL Server. По умолчанию SQL Server *не* выделяет весь этот объем при запуске. Однако после того, как достигнуто это пороговое значение, объем выделенной памяти уже не сможет стать ниже его.

Настройки по умолчанию позволяют SQL Server выделять всю доступную память, не резервируя память для экземпляра. Это приемлемо для многих систем с низкой или даже средней нагрузкой. Но в вашей среде, возможно, будет лучше перенастроить эти параметры (особенно Maximum Server Memory). При этом имейте в виду, что неправильная конфигурация памяти может навредить системе больше, чем если бы вы просто оставили значения по умолчанию.

Параметру Maximum Server Memory обычно нужна настройка. Можно начать с базового значения, а затем отрегулировать его, отслеживая доступную физическую память на сервере с помощью столбца `available_physical_memory_kb` в представлении `sys.dm_os_sys_memory` или с помощью счетчика производительности `Memory\Available MBytes`. Оставьте зарезервированными не менее 512 Мбайт памяти на небольших серверах и не менее 1 Гбайт на серверах со 128 Гбайт ОЗУ или более.

Базовое значение можно рассчитать по формуле:

`общая_физическая_память` – (4 Гбайт + 1 Гбайт × (`общая_физическая_память` – – 16 Гбайт) / 8) – `память_для_других_приложений`

Если вы работаете с SQL Server старше версии 2012, зарезервируйте дополнительную память, потому что в этих версиях параметр Maximum Server Memory отвечает за использование памяти только для буферного пула.

Я не рассматриваю конфигурацию памяти в 32-разрядных версиях SQL Server. Если вы все еще используете их, то самое время обновиться!

Важно правильно оценить, сколько памяти требуется другим приложениям, и зарезервировать часть памяти для них. Сторонние приложения усложняют управление памятью и могут влиять на производительность системы. Для критически важных систем я настоятельно рекомендую использовать выделенные SQL-серверы и не запускать на них никаких приложений (включая SSIS, SSRS и SSAS).

Параметр Maximum Server Memory не мешает SQL Server реагировать на нехватку памяти. В некоторых крайних случаях Windows может даже выгружать часть физической памяти SQL Server в файл подкачки. Это можно предотвратить, если предоставить SQL Server разрешение *Lock Pages In Memory* (*Блокировка страниц в памяти*), сокращенно *LPIM*, в групповой политике.

Разрешение LPIM может улучшить отзывчивость SQL Server в случае экстремальной нехватки внешней памяти. Но с этой настройкой нужно обращаться очень осторожно, особенно в невыделенных средах. Она требует грамотно регулировать максимальную память сервера и может привести к проблемам со стабильностью ОС и даже к отказам, если выделять слишком много памяти. Тем не менее я рекомендую включать LPIM и правильно настраивать параметры памяти, когда это возможно.

На больших серверах из-за накладных расходов на управление памятью также может увеличиться время завершения работы, а оно влияет на продолжительность аварийного переключения в отказоустойчивом кластере SQL Server. С этой проблемой можно справиться, если включить режим *выделения больших страниц* (*large page allocations*) с помощью флага трассировки `T834`. В этом режиме SQL Server выделяет память бóльшими порциями, что ускоряет процесс и снижает накладные расходы. Для этой настройки нужно включить LPIM и заставить SQL Server при запуске выделять сразу всю память вплоть до Maximum Server Memory. Это может замедлить запуск SQL Server, особенно на серверах с большим объемом памяти.

Прежде чем включать режим выделения больших страниц, изучите, как он работает. Вряд ли он поможет, если на сервере нет хотя бы 384 Гбайт оперативной памяти. Не включайте его в невыделенных средах или в средах, использующих индексы columnstore. К сожалению, выделение больших страниц и columnstore плохо работают вместе.

Эта проблема частично решена в SQL Server 2019, где некоторые функции выделения больших страниц можно использовать в средах с индексами columnstore. Для этого надо использовать флаг трассировки `T876` вместо `T834`. Но конфигурацию все равно стоит тщательно протестировать, прежде чем запускать систему.

Сколько нужно памяти?

Про оборудование мы уже говорили в главе 1, но кое-что следует повторить. Память — самый важный ресурс в SQL Server. *Добавить дополнительную память — это зачастую самый быстрый и дешевый способ улучшить производительность системы.*

В Enterprise Edition не ограничен объем памяти, которым может располагать SQL Server. Подключите столько памяти, сколько поддерживает сервер, причем используйте максимально быструю память. Обращайте внимание на объем активно используемых данных (нет смысла создавать сервер с терабайтами ОЗУ для базы данных объемом 100 Гбайт), но в то же время учитывайте будущий рост и обеспечьте достаточно памяти про запас.

В Standard Edition размер буферного пула ограничен 128 Гбайт, но потребуется также память для других компонентов SQL Server и ОС. Я рекомендую обеспечить как минимум 192 Гбайт ОЗУ, чтобы обезопасить систему.

Вам понадобится еще больше памяти, если вы используете индексы columnstore или In-Memory OLTP. В первом случае для хранения данных о сегментах дополнительно используется по 32 Гбайт на каждый экземпляр Standard Edition. Во втором случае понадобится до 32 Гбайт дополнительной памяти *на каждую базу данных*. Стоит добавить памяти и для других служб SQL Server, таких как SSIS, SSAS и SSRS, если они установлены на этом же оборудовании.

Чтобы оптимизировать потребление памяти, можно уменьшить фрагментацию внутреннего индекса (столбец `avg_page_space_used_in_percent` в представлении `sys.dm_db_index_physical_stats`) и/или применить сжатие данных. Это уменьшит количество страниц данных и позволит SQL Server кэшировать больше данных в буферном пуле.

Не могу не подчеркнуть: память сейчас дешевая. Пользуйтесь!

Выделение памяти

Как я упоминал в главе 2, выделение памяти в SQL Server почти всегда происходит через SQLOS. Расширенные хранимые процедуры и поставщики связанных серверов могут выделять память вне SQLOS, и, следовательно, это выделение не будет контролироваться параметром Maximum Server Memory.

На внутреннем уровне SQLOS разбивает память на *узлы памяти* в соответствии с конфигурацией неоднородного доступа к памяти (NUMA): один узел памяти на узел NUMA. У каждого узла памяти есть *распределитель (memory allocator)*, который обращается к различным методам API Windows и Linux, чтобы выделять и освобождать память.

В прежних версиях SQL Server для выделения памяти менее 8 Кбайт применялся одностраничный распределитель, а для выделения более 8 Кбайт — многостраничный. Начиная с SQL Server 2012, распределители памяти объединены в один распределитель *страниц любого размера*. Чтобы отслеживать использование и выделение памяти для каждого узла памяти, можно использовать представление `sys.dm_os_memory_nodes`[1].

На внутреннем уровне каждое выделение памяти становится *объектом памяти*. Каждый объект содержит выделенную память и ее метаданные (размер, сведения о владельце и т. д.). Объекты памяти можно анализировать с помощью представления `sys.dm_os_memory_objects`[2], хотя я им редко пользуюсь.

Еще один ключевой элемент архитектуры памяти SQL Server — *клерки памяти (memory clerks)*. У каждого основного компонента SQL Server есть собственный клерк памяти, который работает как посредник между компонентом и распределителем памяти. Когда компоненту нужна память, он отправляет запрос соответствующему клерку, который, в свою очередь, получает объект памяти от распределителя.

Последний важный компонент в управлении динамической памятью SQL Server называется *брокером памяти (memory broker)*. Брокеры памяти контролируют клерков, регулируя использование ими памяти в зависимости от доступной памяти процесса, нехватки памяти и других условий. Брокеры памяти не выделяют и не освобождают память сами по себе, но могут посылать клеркам сигнал, чтобы те уменьшили или увеличили свои запросы. В представлении `sys.dm_os_memory_brokers`[3] можно увидеть состояние брокеров памяти и соответствующий объем выделяемой памяти.

На рис. 7.1 показано, как связаны между собой компоненты управления памятью в SQL Server.

Анализируя использование памяти, я обычно начинаю с просмотра клерков памяти в представлении `sys.dm_os_memory_clerks`[4]. В листинге 7.3 показано, как это сделать. Этот код будет работать в SQL Server 2012 и более поздних версиях. В более старых версиях нужно заменить столбец `single_page_kb` на сумму столбцов `pages_kb` и `multi_pages_kb`.

---

[1] https://oreil.ly/bWFG8

[2] https://oreil.ly/yZYl2

[3] https://oreil.ly/jDoBP

[4] https://oreil.ly/i1msB

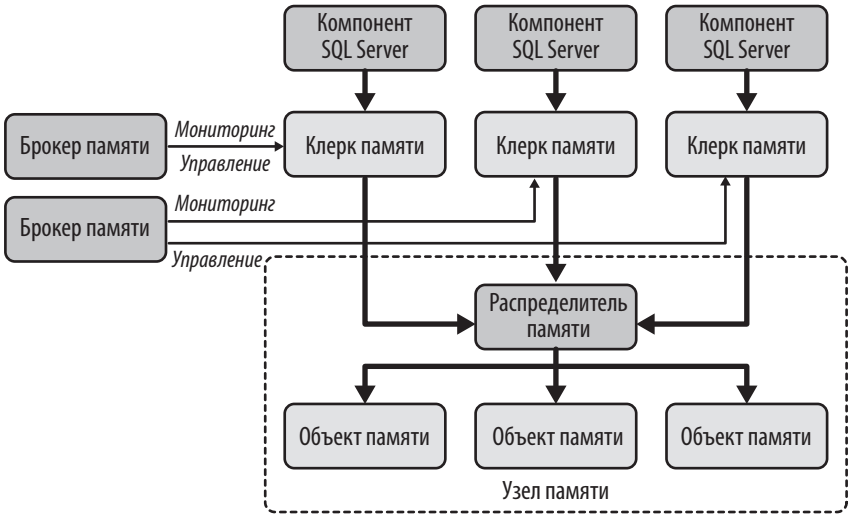


Рис. 7.1. Компоненты управления памятью

Листинг 7.3. Анализ использования памяти

```
SELECT TOP 15
    [type] AS [Memory Clerk]
    ,CONVERT(DECIMAL(16,3),SUM(pages_kb) / 1024.0) AS [Memory Usage(MB)]
FROM sys.dm_os_memory_clerks WITH (NOLOCK)
GROUP BY [type]
ORDER BY sum(pages_kb) DESC;
```

На рис. 7.2 показан вывод этого сценария на одном из промышленных SQL-серверов с 640 Гбайт ОЗУ. Представление `sys.dm_os_memory_clerks` дает достаточно информации, чтобы быстро оценить использование памяти и обнаружить потенциальные аномалии.

	Memory Clerk	Memory Usage (MB)
1	MEMORYCLERK_SQLBUFFERPOOL	583425.398
2	CACHESTORE_OBJCP	10116.328
3	OBJECTSTORE_LOCK_MANAGER	7620.531
4	CACHESTORE_SQLCP	4388.898
5	MEMORYCLERK_BITMAP	1542.930
6	OBJECTSTORE_XACT_CACHE	1083.602
7	MEMORYCLERK_SOSNODE	534.867
8	OBJECTSTORE_SERVICE_BROKER	423.273
9	MEMORYCLERK_SQLSTORENG	315.492
10	USERSTORE_TOKENPERM	275.320

Рис. 7.2. Использование памяти на одном из промышленных серверов

Клерки памяти

Полный список клерков памяти можно найти в документации Microsoft[1]. Давайте рассмотрим наиболее распространенных клерков, с которыми вы можете столкнуться.

MEMORYCLERK_SQLBUFFERPOOL

Как можно догадаться по имени, `MEMORYCLERK_SQLBUFFERPOOL` управляет распределением памяти в буферном пуле. Обычно это один из самых активных потребителей памяти, особенно когда сервер работает с большими базами данных.

Не существует универсального значения для *идеального* размера буферного пула; все зависит от вашей системы. Большие буферные пулы — это нормально. При этом SQL Server просто кэширует больше данных. Волноваться здесь не о чем: когда другим компонентам SQL Server требуется больше памяти, он сокращает размер пула буферов, если в ОС не осталось доступной памяти.

А вот маленькие буферные пулы заслуживают особого внимания. Они не обязательно создают проблемы: помните, что SQL Server кэширует *активные* данные, которые использует система. Например, даже если у вас большая база данных на несколько терабайт, буферный пул может быть маленьким, когда вы работаете с небольшим подмножеством данных.

Можно изучить счетчик производительности *Page Life Expectancy* (листинг 3.3), количество чтений в файлах данных (листинг 3.1) и процент ожиданий `PAGEIOLATCH`, чтобы определить, не происходит ли постоянная очистка буферного пула. В этом случае, особенно в системах OLTP, проанализируйте использование памяти другими клерками и потенциальные условия нехватки памяти. Очевидно, по-прежнему важно обнаруживать и оптимизировать неэффективные запросы: чем меньше данных просматривает запрос, тем меньше их придется размещать в буферном пуле.

Помните, что в версиях SQL Server, отличных от Enterprise, максимальный размер буферного пула ограничен. Например, версия Standard использует не более 128 Гбайт ОЗУ в SQL Server 2014 и более поздних версиях и не более 64 Гбайт в более старых версиях.

В листинге 7.4 показан запрос, позволяющий узнать, как буферный пул использует память для каждой базы данных. Эти сведения полезны, если на сервере несколько баз данных. Код также возвращает среднее время физического чтения страниц данных с диска.

---

[1] https://oreil.ly/9Vp3k

Листинг 7.4. Использование буферного пула для каждой базы данных

```
;WITH BufPoolStats
AS
(
    SELECT
        database_id
        ,COUNT_BIG(*) AS page_count
        ,CONVERT(DECIMAL(16,3),COUNT_BIG(*) * 8 / 1024.) AS size_mb
        ,AVG(read_microsec) AS avg_read_microsec
    FROM
        sys.dm_os_buffer_descriptors WITH (NOLOCK)
    GROUP BY
        database_id
)
SELECT
    DB_NAME(database_id) AS [DB]
    ,size_mb
    ,page_count
    ,avg_read_microsec
    ,CONVERT(DECIMAL(5,2), 100. * (size_mb / SUM(size_mb) OVER()))
        AS [Percent]
FROM
    BufPoolStats
ORDER BY
    size_mb DESC
OPTION (MAXDOP 1, RECOMPILE);
```

CACHESTORE_OBJCP, CACHESTORE_SQLCP и CACHESTORE_PHDR

Клерки CACHESTORE и USERSTORE — это, по сути, кэши для различных типов данных. Клерки CACHESTORE_OBJCP, CACHESTORE_SQLCP и CACHESTORE_PHDR хранят объекты, связанные с кэшем планов:

CACHESTORE_PHDR

Кэширует внутренние объекты, используемые во время компиляции запросов.

CACHESTORE_OBJCP

Сохраняет скомпилированные планы выполнения для хранимых процедур, функций, триггеров и других объектов SQL.

CACHESTORE_SQLCP

Сохраняет скомпилированные планы выполнения для нерегламентированных запросов, подготовленных инструкций и курсоров на стороне сервера.

Клерки CACHESTORE_PHDR чаще всего используются для компиляции запросов, которые ссылаются на сложные представления, большие пакеты и инструкции

с большим количеством констант в предложении IN. Эти клерки кэшируют объекты на короткий срок и делают это только во время компиляции запроса.

Клерк CACHESTORE_PHDR редко потребляет большие объемы памяти. Если вы видите, что он использует много памяти, проанализируйте код и схему базы данных, чтобы найти возможность для рефакторинга.

В большинстве систем бóльшую часть памяти кэша планов выделяют клерки CACHESTORE_OBJCP и CACHESTORE_SQLCP. Потребление памяти CACHESTORE_OBJCP зависит от схемы базы данных и архитектуры уровня данных. Системы с большим количеством активно используемых хранимых процедур и других модулей T-SQL используют больше памяти под планы выполнения. Высокое потребление памяти CACHESTORE_OBJCP *в разумных пределах* не вызывает проблем, если только не влияет на другие компоненты.

С клерками CACHESTORE_SQLCP дела обстоят иначе. Когда они потребляют много памяти, это обычно указывает на чрезмерную нерегламентированную рабочую нагрузку, которая интенсивно использует ЦП и кэш планов.

В предыдущей главе вы уже видели, какие накладные расходы ЦП возникают из-за нерегламентированных запросов. Теперь рассмотрим пример накладных расходов памяти из-за этих запросов. В листинге 7.5 выполняется 1000 простых нерегламентированных запросов, а затем с помощью представления sys.dm_exec_cached_plans[1] проверяется состояние кэша планов. Этот сценарий очищает содержимое кэша с помощью команды DBCC FREEPROCCACHE — не запускайте его на рабочем сервере!

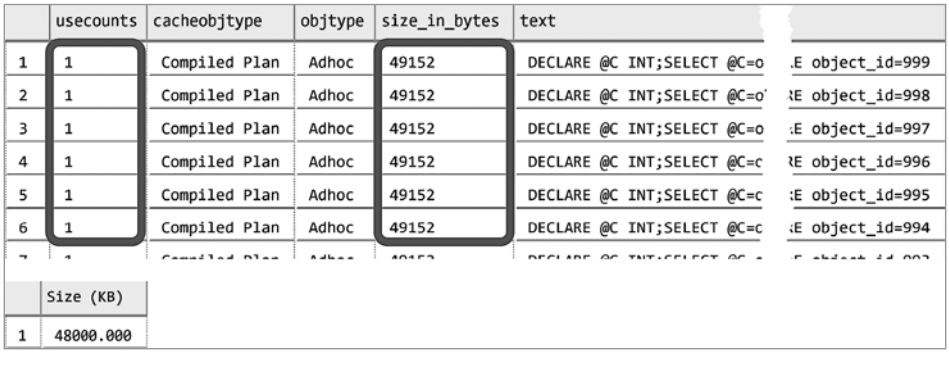
Листинг 7.5. Выполнение 1000 специальных запросов и проверка содержимого кэша планов

```
DBCC FREEPROCCACHE
GO

DECLARE
    @SQL NVARCHAR(MAX)
    ,@I INT = 0

WHILE @I < 1000
BEGIN
    SELECT @SQL =
N'DECLARE @C INT;SELECT @C=object_id FROM sys.objects WHERE object_id='
+ CONVERT(NVARCHAR(10),@I);
    EXEC(@SQL);
    SELECT @I += 1;
END;

SELECT
    p.usecounts, p.cacheobjtype, p.objtype, p.size_in_bytes, t.[text]
```

[1] https://oreil.ly/sNg5A

```
FROM
    sys.dm_exec_cached_plans p WITH (NOLOCK)
        CROSS APPLY sys.dm_exec_sql_text(p.plan_handle) t
WHERE
    p.objtype = 'Adhoc'
ORDER BY
    p.size_in_bytes DESC
OPTION (RECOMPILE);

SELECT
    CONVERT(DECIMAL(12,3),SUM(1. * p.size_in_bytes)/1024.) AS [Size (KB)]
FROM
    sys.dm_exec_cached_plans p WITH (NOLOCK)
WHERE
    p.objtype = 'Adhoc'
OPTION (RECOMPILE);
```

На рис. 7.3 показано содержимое кэша планов и потребление им памяти, когда этот запрос запускается с отключенным параметром конфигурации Optimize for Ad-Hoc Workloads. Кэшируется 1000 планов, каждый из которых использует 48 Кбайт памяти, что дает в сумме 48 Мбайт. Имейте в виду, что это был очень простой запрос. Большие и сложные запросы потребуют значительно больше памяти для планов выполнения.



	usecounts	cacheobjtype	objtype	size_in_bytes	text	
1	1	Compiled Plan	Adhoc	49152	DECLARE @C INT;SELECT @C=o	.E object_id=999
2	1	Compiled Plan	Adhoc	49152	DECLARE @C INT;SELECT @C=o	:E object_id=998
3	1	Compiled Plan	Adhoc	49152	DECLARE @C INT;SELECT @C=o	:E object_id=997
4	1	Compiled Plan	Adhoc	49152	DECLARE @C INT;SELECT @C=c	:E object_id=996
5	1	Compiled Plan	Adhoc	49152	DECLARE @C INT;SELECT @C=c	:E object_id=995
6	1	Compiled Plan	Adhoc	49152	DECLARE @C INT;SELECT @C=c	:E object_id=994

	Size (KB)
1	48000.000

Рис. 7.3. Содержимое кэша планов, когда оптимизация для нерегламентированных рабочих нагрузок отключена

На рис. 7.4 показана статистика кэша планов после повторного теста с включенным параметром Optimize for Ad-Hoc Workloads. В кэше по-прежнему 1000 планов, однако сценарий потребляет гораздо меньше памяти. Вместо реальных скомпилированных планов SQL Server кэширует небольшие 400-байтовые структуры, называемые *заглушками скомпилированных планов (compiled plan stubs)*. Эти заполнители позволяют отслеживать, какие нерегламентированные запросы были выполнены. Когда запрос выполняется во второй раз, SQL Server

заменяет заглушку фактическим скомпилированным планом и повторно использует его в будущем.

	usecounts	cacheobjtype	objtype	size_in_bytes	text
1	1	Compiled Plan Stub	Adhoc	456	DECLARE @C INT;SELECT @C=(
2	1	Compiled Plan Stub	Adhoc	456	DECLARE @C INT;SELECT @C=
3	1	Compiled Plan Stub	Adhoc	456	DECLARE @C INT;SELECT @C=
4	1	Compiled Plan Stub	Adhoc	456	DECLARE @C INT;SELECT @C=(
5	1	Compiled Plan Stub	Adhoc	456	DECLARE @C INT;SELECT @C=(
6	1	Compiled Plan Stub	Adhoc	456	DECLARE @C INT;SELECT @C=

	Size (KB)
1	398.438

Рис. 7.4. Содержимое кэша планов, когда оптимизация для нерегламентированных рабочих нагрузок включена

Как я уже не раз упоминал, параметр Optimize for Ad-Hoc Workloads стоит включать в большинстве систем. Он уменьшит расход памяти на кэш планов и улучшит производительность.

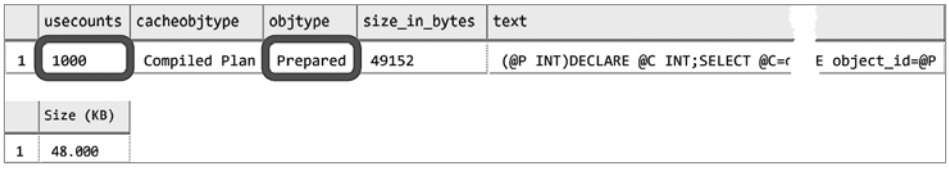
Наконец, повторим тот же тест, используя параметризованный запрос, как показано в листинге 7.6.

Листинг 7.6. Выполнение параметризованного запроса и проверка содержимого кэша плана

```
DBCC FREEPROCCACHE
GO

DECLARE
    @SQL NVARCHAR(MAX)
    ,@I INT = 0

WHILE @I < 1000
BEGIN
    SELECT @SQL =
N'DECLARE @C INT;SELECT @C=object_id FROM sys.objects WHERE object_id=@P';
    EXEC sp_executesql @SQL=@SQL,@Params=N'@P INT',@P = @I;
    SELECT @I += 1;
END;

SELECT
    p.usecounts, p.cacheobjtype, p.objtype, p.size_in_bytes, t.[text]
FROM
    sys.dm_exec_cached_plans p WITH (NOLOCK)
        CROSS APPLY sys.dm_exec_sql_text(p.plan_handle) t
WHERE
    p.objtype = 'Prepared'
```

```
ORDER BY
    p.size_in_bytes DESC
OPTION (RECOMPILE);

SELECT
    CONVERT(DECIMAL(12,3),SUM(1. * p.size_in_bytes)/1024.) AS [Size (KB)]
FROM
    sys.dm_exec_cached_plans p WITH (NOLOCK)
WHERE
    p.objtype = 'Prepared'
OPTION (RECOMPILE);
```

Как видно на рис. 7.5, кэшируется всего один план выполнения. Это снижает потребление памяти и нагрузку на ЦП.



	usecounts	cacheobjtype	objtype	size_in_bytes	text
1	1000	Compiled Plan	Prepared	49152	(@P INT)DECLARE @C INT;SELECT @C=c E object_id=@P

	Size (KB)
1	48.000

Рис. 7.5. Содержимое кэша планов в случае параметризованного запроса

Объем памяти, занятый кэшем планов, зависит от рабочей нагрузки в системе, сложности планов выполнения и структуры уровня данных. Нередко клерки памяти потребляют гигабайты или даже десятки гигабайт памяти. Однако в любом случае следите за этим потреблением и изучайте содержимое кэша планов, если видите большие цифры.

Это особенно важно, когда много памяти потребляет клерк `CACHESTORE_SQLCP`. Чаще всего это признак большой нерегламентированной нагрузки и большого количества одноразовых планов. Как уже отмечалось в этой главе, это влияет на производительность системы.

Существует еще один важный момент, касающийся нерегламентированной рабочей нагрузки. По умолчанию SQL Server может хранить в кэше планов около 160 000 объектов. Если достигнут этот предел, возможны дополнительные состязания за ресурсы ЦП.

Емкость кэша можно увеличить примерно до 640 000 объектов, если включить флаг трассировки `T174`. Это может уменьшить загрузку ЦП в системах с очень большими нерегламентированными рабочими нагрузками, но также может ухудшить производительность, потому что SQL Server начнет использовать больше памяти для кэша планов. Протестируйте эту настройку перед тем, как развертывать в рабочей среде.

В листинге 7.7 показаны три запроса, с помощью которых можно анализировать кэш планов и устранять неполадки. Первый запрос возвращает информацию

о кэшированных объектах кэша планов, группируя объекты по типам. Данные собираются от всех клерков памяти, обращающихся к кэшу планов.

Второй запрос выводит количество одноразовых планов и объем памяти, который они занимают в кэше. Большое количество планов и особенно большое потребление памяти стоит изучить подробнее. Третий запрос помогает обнаружить одноразовые планы с наиболее интенсивным потреблением памяти.

Листинг 7.7. Анализ кэша планов

```
-- Количество кэшированных объектов и использование ими памяти с группировкой
по типам
SELECT
    cacheobjtype
    ,objtype
    ,COUNT(*) AS [Count]
    ,CONVERT(DECIMAL(15,3)
        ,SUM(CONVERT(BIGINT,size_in_bytes))/1024./1024.)
            AS [Size (MB)]
FROM
    sys.dm_exec_cached_plans WITH (NOLOCK)
GROUP BY
    cacheobjtype, objtype
ORDER BY
    [Size (MB)] DESC
OPTION (RECOMPILE);

-- Статистика одноразовых планов выполнения
SELECT
    COUNT(*) AS [Single-used plan count]
    ,CONVERT(DECIMAL(10,3)
        ,SUM(CONVERT(BIGINT,cp.size_in_bytes))/1024./1024.)
            AS [Size (MB)]
FROM
    sys.dm_exec_cached_plans cp WITH (NOLOCK)
WHERE
    cp.objtype in (N'Adhoc', N'Prepared') AND
    cp.usecounts = 1
OPTION (RECOMPILE);

-- 25 одноразовых планов с наиболее интенсивным потреблением памяти
SELECT TOP 25
    DB_NAME(t.dbid) as [DB]
    ,cp.usecounts
    ,cp.plan_handle
    ,t.[text]
    ,cp.objtype
    ,cp.size_in_bytes
    ,CONVERT(DECIMAL(12,3),cp.size_in_bytes/1024.) as [Size (KB)]
FROM
    sys.dm_exec_cached_plans cp WITH (NOLOCK)
        CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) t
```

```
WHERE
    cp.cacheobjtype = N'Compiled Plan'
    AND cp.objtype in (N'Adhoc', N'Prepared')
    AND cp.usecounts = 1
ORDER BY
    cp.size_in_bytes DESC
OPTION (RECOMPILE);
```

Чтобы удалить из кэша отдельные планы выполнения, можно вызвать инструкцию `DBCC FREEPROCCACHE` и передать дескриптор плана в качестве параметра. Это может быть полезно, если вы хотите удалить конкретные большие одноразовые планы, не трогая другие. Таким же образом можно удалить регрессивные планы, чувствительные к параметрам и затронутые сканированием параметров.

Чтобы удалить все планы, сохраненные клерком `CACHESTORE_SQLCP`, используйте команду `DBCC FREESYSTEMCACHE('SQL Plans') WITH MARK_IN_USE_FOR_REMOVAL`. Она удалит все нерегламентированные и подготовленные планы из кэша независимо от того, как часто они использовались повторно, а планы из хранимых процедур и других модулей T-SQL останутся нетронутыми.

Очевидно, по возможности стоит устранить первопричину проблемы. Включите параметр Optimize for Ad-Hoc Workloads и параметризуйте запросы, как описано в главе 6.

Также следует насторожиться, если кэш планов невелик и занимает слишком мало памяти. Это может быть нормально, когда в системе используются параметризованные запросы и хранимые процедуры, а нерегламентированной активности мало. В то же время, если кэш планов постоянно сокращается, это может говорить о нехватке памяти. В этом случае проанализируйте счетчики производительности компиляции и перекомпиляции.

OBJECTSTORE_LOCK_MANAGER

Клерк `OBJECTSTORE_LOCK_MANAGER` хранит структуры блокировок, которые SQL Server использует для поддержки конкурентного доступа. Если этот клерк потребляет много памяти, это указывает на большое количество активных блокировок. Обычно количество активных блокировок стоит сводить к минимуму, но бывают случаи, когда необходимо установить много блокировок на уровне строк и предотвратить их распространение на уровень объектов.

В главе 8 я расскажу о том, как устранять проблемы конкурентного доступа в SQL Server.

MEMORYCLERK_SQLQERESERVATIONS

Клерк `MEMORYCLERK_SQLQERESERVATIONS` управляет *предоставлением памяти* — выделением памяти запросам во время их выполнения. Этот клерк часто бы-

вает одним из основных потребителей памяти, особенно в хранилищах данных и системах с отчетной нагрузкой.

Позже в этой главе я расскажу об управлении памятью во время выполнения запросов и об устранении неполадок, связанных с избыточным предоставлением памяти.

USERSTORE_TOKENPERM

Клерк `USERSTORE_TOKENPERM` выделяет память для хранилища маркеров безопасности, которое используется, чтобы отслеживать разрешения пользователей и многие другие объекты безопасности. Большое хранилище маркеров может вызвать проблемы с производительностью, когда увеличивается нагрузка на ЦП и возникает нехватка внутренней памяти из-за того, что хранилище заимствует ее у других компонентов SQL Server.

К сожалению, с этой неприятностью тяжело справиться. Что еще хуже, с хранилищем маркеров в SQL Server связано много известных проблем. Некоторые из них происходят из-за интенсивного использования ролей приложений, а также из-за большой нерегламентированной рабочей нагрузки.

В общем, за клерком памяти `USERSTORE_TOKENPERM` нужно приглядывать. Если он занимает больше нескольких гигабайт, это может быть тревожным сигналом, особенно когда потребление памяти продолжает расти. В этом случае установите последний пакет обновления и/или накопительное обновление. Если это не поможет, попробуйте обратиться в службу поддержки Microsoft.

В качестве временного решения можно очистить хранилище маркеров командой `DBCC FREESYS TEMCACHE('TokenAndPermUserStore')`. В некоторых случаях может понадобиться регулярно очищать хранилище с помощью задания агента SQL Server, пока не найдется окончательное решение.

MEMORYCLERK_SQLCONNECTIONPOOL

Клерк `MEMORYCLERK_SQLCONNECTIONPOOL` предоставляет память для объектов, связанных с подключением клиентов к серверу. Самый типичный пример таких объектов — дескрипторы подготовленных инструкций, созданные вызовами хранимой процедуры `sp_prepexecrpc` во время некоторых вызовов удаленных процедур.

Этот клерк редко потребляет много памяти, но его аппетит может увеличиться, если клиентские приложения не закрывают подготовленные дескрипторы корректно, поддерживая открытые соединения с базой данных. Чтобы очистить память, вам, возможно, придется перезапустить приложение, а в некоторых случаях даже SQL Server.

Очевидно, лучше устранить основную причину проблемы в коде приложения и закрывать дескрипторы после выполнения.

MEMORYCLERK_SQLCLR, MEMORYCLERK_SQLCLRASSEMBLY и MEMORYCLERK_SQLEXTENSIBILITY

Клерки `MEMORYCLERK_SQLCLR`, `MEMORYCLERK_SQLCLRASSEMBLY` и `MEMORYCLERK_SQLEXTENSIBILITY` используются для выделения памяти в общеязыковой среде выполнения (CLR) и для поддержки языковых расширений (Java, R и Python). Все внешние языки умеют управлять памятью и автоматически выполнять сборку мусора, но можно написать код, который потребляет много памяти во время выполнения — например, когда обрабатываются большие файлы.

Если эти клерки используют большой объем памяти, присмотритесь к CLR и/или внешним языкам. Возможно, придется проконсультироваться с разработчиками, провести рефакторинг или даже перенести часть кода на серверы приложений.

MEMORYCLERK_XTP

Клерк `MEMORYCLERK_XTP` управляет выделением памяти для технологии In-Memory OLTP. Когда он использует много памяти, нужно посмотреть, как ее потребляют таблицы, оптимизированные для памяти, а также изучить некоторые другие вещи, о которых я скажу позже в этой главе.

Для In-Memory OLTP высокое потребление памяти может быть вполне нормально, когда в таблицах, оптимизированных для памяти, хранятся большие объемы данных. Что еще более важно, эта память не будет высвобождаться при нехватке памяти, пока используется In-Memory OLTP. Это следует учитывать, планируя мощность оборудования для серверов: вторичные реплики групп доступности, серверы аварийного восстановления, резервные среды и т. д. Если на сервере недостаточно памяти, чтобы разместить данные In-Memory OLTP, то база данных не запустится и/или данные в таблицах, оптимизированных для памяти, окажутся доступны только для чтения.

Подведем итог

В этом разделе невозможно охватить, да и не нужно охватывать всех клерков памяти. К счастью, в документации Microsoft[1] есть их исчерпывающий список. Хотя документация не рассказывает, как устранять неполадки, она все же даст представление о том, за что отвечает каждый клерк памяти, и направит вас в нужную сторону.

Избегайте однобокого взгляда на проблему и не делайте поспешных выводов из одного лишь потребления памяти клерками. За очень редкими исключениями, не существует конкретных нормативов по тому, сколько памяти должен использовать тот или иной клерк. Стоит рассматривать потребление памяти целостно и понимать, как разные компоненты влияют друг на друга.

---

[1] https://oreil.ly/Q3gF2

Например, когда для выполнения запроса резервируется много памяти (MEMORYCLERK_SQLQERESERVATIONS), это может быть совершенно нормальным, если не влияет на буферный пул, кэш планов и другие компоненты SQL Server. Вместе с тем такое резервирование может навредить, если из-за него возникает нехватка внутренней памяти. Анализируя систему, сопоставляйте информацию о клерках памяти с другими метриками.

Команда DBCC MEMORYSTATUS

Если вы много работали с SQL Server, то, вероятно, знакомы с командой DBCC MEMORYSTATUS, которая предоставляет моментальный снимок текущего потребления памяти в SQL Server. По моему мнению, в этой команде много чего намешано. Она объединяет всю информацию об использовании памяти в один вывод из нескольких наборов результатов, но с ней почти не удается фильтровать и агрегировать данные. Поэтому я чаще собираю метрики с помощью динамических административных представлений.

В этой книге я не буду рассматривать команду DBCC MEMORYSTATUS, но советую запустить ее, чтобы посмотреть, как она представляет информацию и легко ли ее интерпретировать. Это всего лишь еще одна консолидированная проекция данных, о которых уже говорилось в этой главе.

Выполнение запросов и предоставление памяти

Любому запросу в SQL Server для выполнения нужна память. Эта *предоставляемая память* выделяется запросу перед выполнением. Запрос не запустится, пока не получит память, а если он долго не запускается, то его время ожидания может истечь.

Размер предоставляемой памяти зависит от операторов в плане выполнения, оценки количества элементов, степени параллелизма, режима выполнения (построчный или пакетный) и некоторых других факторов. Например, операторам *Sort* или *Hash* нужна отдельная память для создания внутренних структур. Им также следует выделить дополнительную память, чтобы хранить все данные или их подмножество в памяти, а не переносить их в tempdb.

Информацию о предоставлении памяти можно увидеть в плане выполнения запроса. На рис. 7.6 показана информация, доступная в окне плана запроса во всплывающем окне SELECT (верхнего оператора в плане) и в окне свойств SSMS. Те же данные можно получить из XML-представления плана выполнения. Значения выражены в килобайтах.

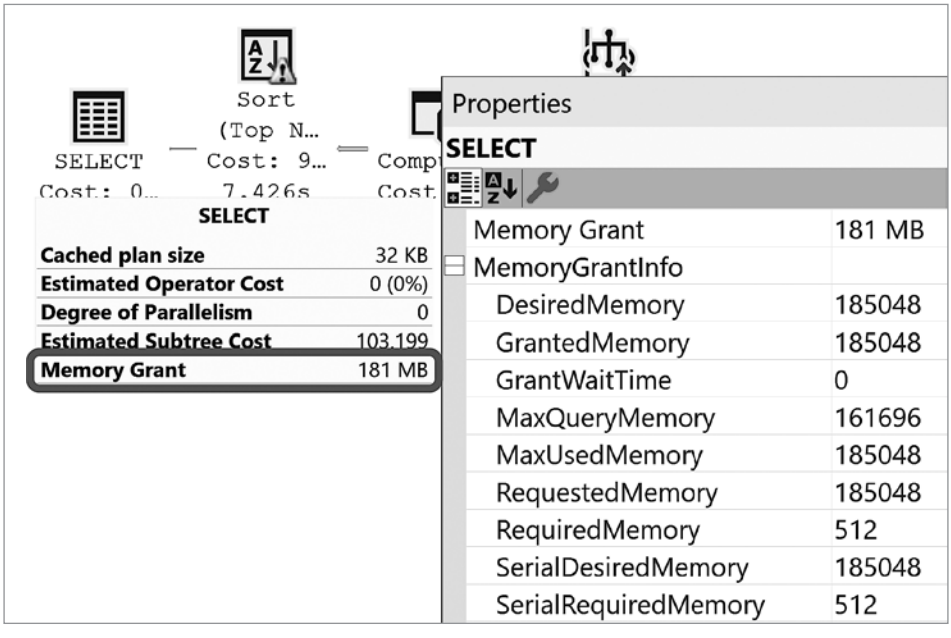


Рис. 7.6. Информация о предоставлении памяти в SSMS

Давайте рассмотрим свойства предоставления памяти.

`RequiredMemory`

Абсолютный минимальный объем памяти, необходимый для выполнения запроса. Запрос не запустится, пока у него не будет хотя бы этого объема памяти.

`SerialRequiredMemory`

Абсолютный минимальный объем памяти, необходимый для выполнения запроса в случае последовательного плана выполнения. Равен `RequiredMemory`, если запрос выполняется последовательно.

`DesiredMemory`

Объем памяти, нужной запросу в идеальной ситуации. Например, если в плане запроса есть оператор *Sort*, то `DesiredMemory` может предусмотреть достаточно памяти, чтобы сортировать в ней все данные на основе оценки количества элементов.

`SerialDesiredMemory`

Желаемый объем памяти, если запрос выполняется последовательно.

`RequestedMemory`

Объем памяти, который запрос потребовал от SQL Server, обращаясь за предоставлением памяти.

`Granted`

Объем памяти, предоставленной запросу.

`MaxUsedMemory`

Объем памяти, которую запрос использует во время выполнения.

`MaxQueryMemory`

Максимально возможный объем предоставленной памяти для запроса.

`GrantWaitTime`

Количество времени, в течение которого запрос ожидал предоставления памяти.

Объем предоставляемой памяти рассчитывается во время оптимизации запроса и кэшируется вместе с планом выполнения. При повторных выполнениях запроса будет предоставляться такой же объем памяти, однако SQL Server 2017 и более поздние версии могут пересчитывать размер предоставляемой памяти, исходя из ее фактического использования в предыдущих выполнениях. Я расскажу об этой функции позже в этой главе.

Как я упоминал ранее, предоставлением памяти управляет клерк `MEMORYCLERK_SQLQERESERVATION`, который использует для выделения памяти объект синхронизации потоков, называемый *семафором ресурсов (resource semaphore)*.

Когда выделить память не удается, семафор ресурсов помещает запросы в очередь ожидания, создавая ожидания `RESOURCE_SEMAPHORE`. На внутреннем уровне семафор использует две очереди ожидания и ранжирует запросы по объему предоставляемой памяти и по стоимости запроса. Первая очередь, называемая *семафором ресурсов для небольших запросов*, хранит запросы, для которых требуется менее 5 Мбайт памяти и стоимость которых составляет менее трех секунд. Во второй очереди хранятся все остальные запросы.

Семафор ресурсов обрабатывает запросы в порядке очереди. Он отдает предпочтение очереди ресурсов с небольшими запросами и сокращает время ожидания для небольших запросов, не требующих много памяти.

Важно отслеживать запросы, которые ожидают предоставления памяти. Такие ситуации ненормальны и требуют внимания. Большое количество ожиданий `RESOURCE_SEMAPHORE` — признак проблемы.

В объекте `Memory Management` есть несколько счетчиков производительности, которые можно использовать для мониторинга и устранения неполадок.

`Memory Grants Pending`

Количество заявок на предоставление памяти, которые ожидают обработки. В идеале этот счетчик должен быть всегда равен 0: это значит, что ни один запрос не ждет предоставления памяти.

`Memory Grants Outstanding`

Количество выполняющихся в данный момент запросов, которым уже предоставлена память. Большие значения счетчика означают интенсивную нагрузку на память (обычно ее вызывают запросы с операторами *Sort* и *Hash*). В хранилищах данных это нормально, но в системах OLTP эту ситуацию нужно исследовать.

`Maximum Workspace Memory`

Общий объем памяти рабочей области (в килобайтах).

`Granted Workspace Memory`

Объем памяти рабочей области (в килобайтах), который используется в данный момент.

Более подробную информацию об общем и предоставленном объеме памяти рабочей области можно получить с помощью представления `sys.dm_exec_query_resource_semaphores`[1]. В нем отображается статистика обеих очередей семафоров ресурсов, включая сведения о памяти, количество запросов в очереди ожидания и некоторые другие показатели.

Получить информацию о запросах, которые ожидают предоставления памяти или уже выполняются, можно в представлении `sys.dm_exec_query_memory_grants`[2], как показано в листинге 7.8. В столбце `grant_time` отображается время, когда память была предоставлена. Значение `NULL` в этом столбце указывает, что заявка на предоставление памяти ожидает обработки.

Листинг 7.8. Информация о предоставлении памяти

```
SELECT
    mg.session_id
    ,t.text AS [sql]
    ,qp.query_plan AS [plan]
    ,mg.is_small /* Информация очереди семафора ресурсов */
    ,mg.dop
    ,mg.query_cost
    ,mg.request_time
    ,mg.grant_time
    ,mg.wait_time_ms
```

[1] https://oreil.ly/9SuUC

[2] https://oreil.ly/YRSnq

```
    ,mg.required_memory_kb
    ,mg.requested_memory_kb
    ,mg.granted_memory_kb
    ,mg.used_memory_kb
    ,mg.max_used_memory_kb
    ,mg.ideal_memory_kb
FROM
    sys.dm_exec_query_memory_grants mg WITH (NOLOCK)
        CROSS APPLY sys.dm_exec_sql_text(mg.sql_handle) t
        CROSS APPLY sys.dm_exec_query_plan(mg.plan_handle) qp
--WHERE -- Раскомментируйте, чтобы видеть только ожидающие запросы
--     mg.grant_time IS NULL
ORDER BY
    mg.requested_memory_kb DESC
OPTION (RECOMPILE, MAXDOP 1);
```

В представлении `sys.dm_exec_query_memory_grants` показан текущий статус предоставления памяти. Однако вам может понадобиться также просмотреть историю использования памяти и выявить запросы, которые потребляют ее особенно интенсивно. Чтобы анализировать данные о предоставлении памяти, можно использовать методы из главы 4.

Вероятно, проще всего использовать статистику выполнения запросов и представление `sys.dm_exec_query_stats`. В нем есть несколько столбцов, по которым можно отслеживать потребление памяти и ее предоставление запросам. Можно использовать код из листинга 4.1, где отсортировать данные по столбцам `total_grant_kb` и/или `[avg grant kb]` и обнаружить запросы, лидирующие по потреблению памяти. Кроме того, можно получить данные из хранилища запросов, когда оно включено.

Наконец, можно отслеживать заявки на предоставление памяти во время выполнения с помощью расширенных событий. В SQL Server 2014 SP2 и более поздних версиях можно использовать упрощенное профилирование запросов, о котором я говорил в главе 5. В более старых версиях вместо этого можно использовать событие `query_memory_grant_usage`.

Оптимизация запросов, интенсивно использующих память

Обнаружив множество запросов, которые интенсивно используют память, проанализируйте планы их выполнения. В большинстве случаев предоставление больших объемов памяти связано с операторами *Hash* и *Sort*. По возможности получите фактические планы выполнения, потому что в них указано, сколько строк было обработано операторами и как они использовали память.

К сожалению, нет никакого волшебного средства, которое само оптимизировало бы запросы и сокращало потребление ими памяти. Но немного «магии» можно

сотворить своими силами. Прежде всего посмотрите, можно ли улучшить индексацию. От этого прекратятся ненужные сортировки, а в некоторых случаях хеш-соединения заменятся на соединения в цикле, которые потребляют гораздо меньше памяти.

Рассмотрим очередной пример. Код в листинге 7.9 создает таблицу и заполняет ее данными.

Листинг 7.9. Оптимизация запросов, интенсивно использующих память: создание таблицы

```
CREATE TABLE dbo.Orders
(
    OrderID INT NOT NULL,
    OrderDate DATETIME2(0) NOT NULL,
    Placeholder CHAR(8000) NULL,
    CONSTRAINT PK_Orders PRIMARY KEY CLUSTERED(OrderID)
);

;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 строки
,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 строки
,N3(C) AS (SELECT 0 FROM N2 AS T1 CROSS JOIN N2 AS T2) -- 16 строк
,N4(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N3 AS T2) -- 256 строк
,N5(C) AS (SELECT 0 FROM N4 AS T1 CROSS JOIN N4 AS T2) -- 65 536 строк
,IDs(ID) AS (SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) FROM N5)
INSERT INTO dbo.Orders(OrderID, OrderDate)
    SELECT ID, DATEADD(day,ID % 365, '2021-01-01')
    FROM IDs;
```

Теперь запустим запрос, который возвращает 200 самых последних заказов, как показано в листинге 7.10.

Листинг 7.10. Оптимизация запросов, интенсивно использующих память: тестовый запрос 1

```
SELECT TOP 200 OrderID, OrderDate, Placeholder
FROM dbo.Orders
ORDER BY OrderDate DESC
```

Метрики плана выполнения и предоставления памяти для запроса показаны на рис. 7.7. Запросу предоставлена память объемом 630 Мбайт, которая нужна оператору `Sort TOP N`. Он кэширует строки в памяти перед сортировкой.

Стоит отметить, что на внутреннем уровне оператор `Sort TOP N` применяет недокументированную оптимизацию в случаях, когда условие `TOP` не превышает 100 строк. В этом режиме оператор использует очень мало памяти во время выполнения.

Как я уже упоминал, часто удается избежать сортировки с помощью грамотного индексирования. Например, создадим индекс с помощью команды `CREATE INDEX IDX_Orders_OrderDate ON dbo.Orders(OrderDate)` и снова запустим запрос из листинга 7.10.

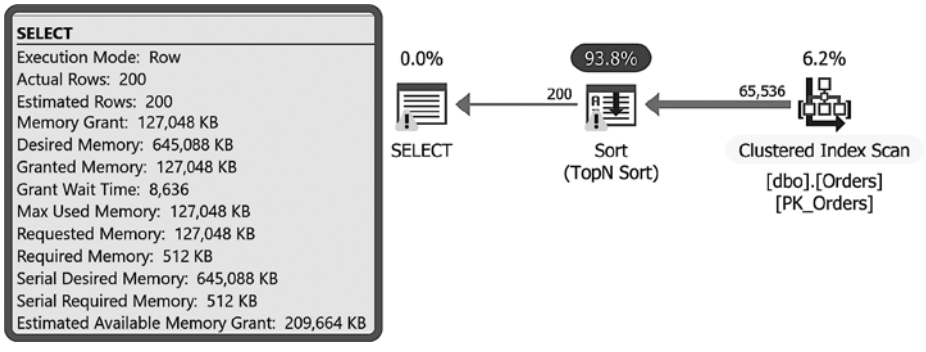


Рис. 7.7. Оптимизация запросов, интенсивно использующих память: план выполнения запроса

На рис. 7.8 показан новый план выполнения. Оператор *Sort* больше не требуется, так что для него не нужно предоставлять память.

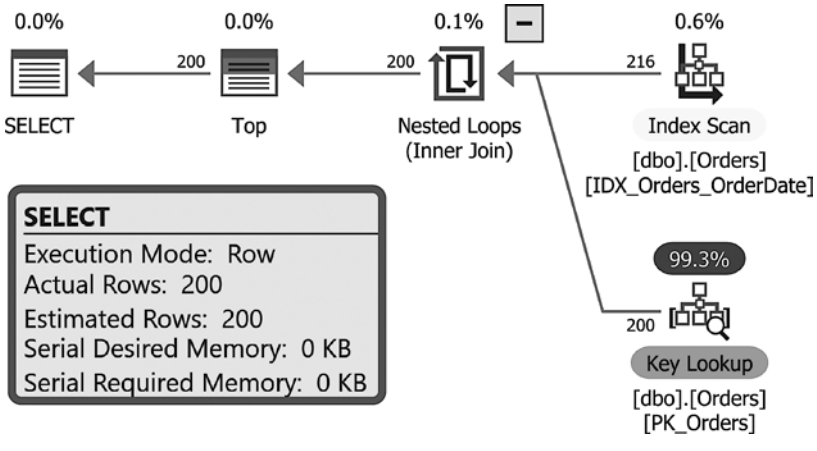


Рис. 7.8. Оптимизация запросов, интенсивно использующих память: план выполнения запроса после создания индекса

Объем запрошенного предоставления памяти зависит от оценочного количества и размера строк, которые оператору нужно обработать. Например, если оптимизатор запросов собирается отсортировать 10 000 строк по 100 байт, то для размещения данных в памяти потребуется около 10 Мбайт. Неверная оценка количества элементов или размера строки может привести к неправильному предоставлению памяти.

Сперва рассмотрим, чем грозит неверная оценка количества элементов, и выполним другой запрос, показанный в листинге 7.11.

Листинг 7.11. Оптимизация запросов, интенсивно использующих память: тестовый запрос 2

```
SELECT TOP 200 OrderID, OrderDate, Placeholder
FROM dbo.Orders
WHERE OrderDate BETWEEN '2021-07-01' AND '2021-08-01'
ORDER BY Placeholder;
```

На рис. 7.9 частично показаны план выполнения запроса, а также статистика предоставления памяти. В этом примере индекс только что создан, поэтому статистика актуальна.

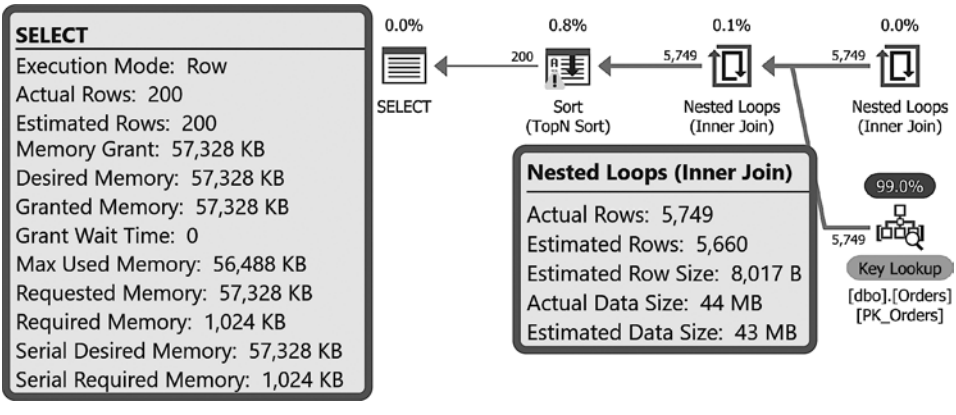


Рис. 7.9. Оптимизация запросов, интенсивно использующих память: план выполнения с актуальной статистикой

Теперь запустим код из листинга 7.12. Он отключает автоматическое обновление статистики в индексе, а затем удаляет множество строк и очищает кэш планов.

Листинг 7.12. Оптимизация запросов, интенсивно использующих память: моделирование устаревшей статистики

```
ALTER INDEX IDX_Orders_OrderDate ON dbo.Orders
SET (STATISTICS_NORECOMPUTE = ON);

DELETE FROM dbo.Orders
WHERE OrderDate BETWEEN '2021-07-02' AND '2021-09-01';

DBCC FREEPROCCACHE;
```

Теперь повторим тест и снова запустим запрос из листинга 7.11. На рис. 7.10 видно, что ошибочная оценка количества элементов привела к неправильному и избыточному предоставлению памяти для запроса.

Обновим статистику командой `UPDATE STATISTICS dbo.Orders IDX_Orders_OrderDate WITH FULLSCAN` и опять запустим запрос. Как видно на рис. 7.11, при правильной оценке количества элементов запросу предоставляется гораздо меньше памяти.

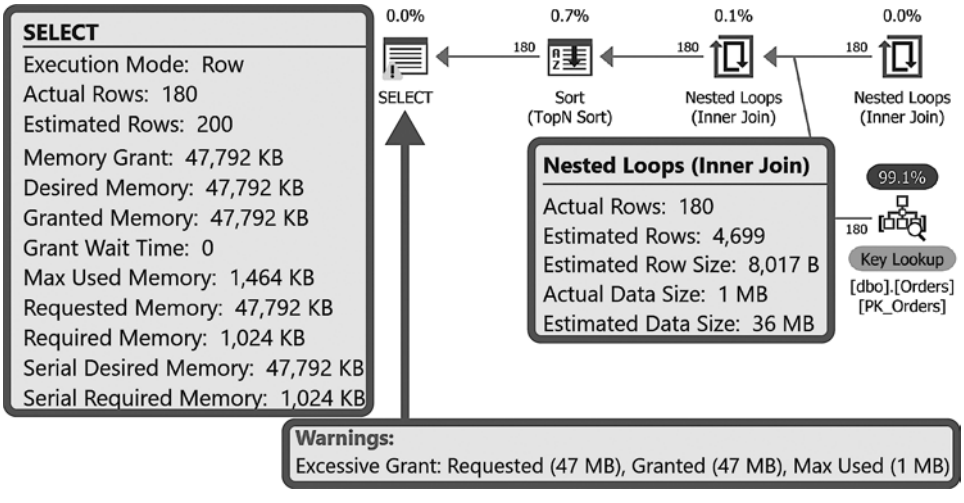


Рис. 7.10. Оптимизация запросов, интенсивно использующих память: неверная оценка количества элементов и предоставление памяти

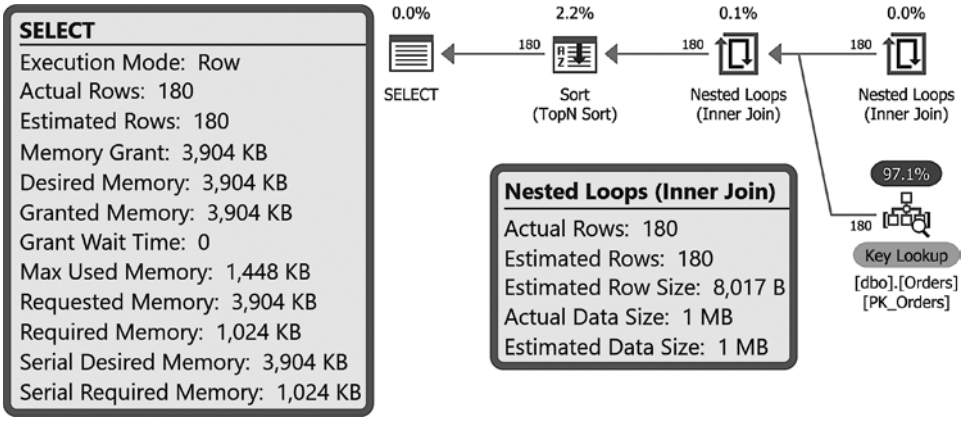


Рис. 7.11. Оптимизация запросов, интенсивно использующих память: предоставление памяти после инструкции `UPDATE STATISTICS`

Разбираться с неверной оценкой количества элементов — трудная задача. Убедитесь, что статистика актуальна, и избегайте любых конструкций, которые могут повлиять на оценку количества элементов (табличные переменные, функции с табличными значениями из нескольких инструкций и т. д.). В некоторых случаях стоит разбивать сложные запросы на более мелкие и сохранять промежуточные результаты во временных таблицах. Это влечет за собой накладные расходы, о которых я расскажу в главе 9, однако порой это приемлемая цена за более эффективные планы выполнения.

Наконец, рассмотрим еще один фактор, от которого зависит объем предоставляемой памяти, — размер строки данных. SQL Server вычисляет его на основе типов данных в столбцах, обрабатываемых операторами. Для столбцов фиксированной длины размер известен заранее. Например, `TINYINT` занимает 1 байт, `INT` — 4 байта и т. д.

С другой стороны, оценки для столбцов переменной длины зависят от того, какая длина указана в определении таблицы. SQL Server оценивает их заполнение в 50 %. Например, столбец, определенный как `VARCHAR(100)`, будет оценен в 50 байт, а `NVARCHAR(200)` — в 200 байт, потому что каждый символ Unicode занимает 2 байта[1]. Наконец, столбцы, определенные как `(MAX)`, будут оценены в 4000 байт.

Не выбирайте ненужные столбцы и старайтесь не использовать типы `(N)CHAR(N)` и `BINARY(N)` с большими значениями `N`, потому что при этом увеличивается оценочный размер строки и объем выделяемой памяти. Предыдущие примеры демонстрируют эффект от увесистого столбца `CHAR(8000)`. SQL Server оценил каждую строку данных в 8017 байт, несмотря на то что во всех строках в столбце `Placeholder` было значение `NULL`.

Изменим тип данных столбца `Placeholder` командой `ALTER TABLE dbo.Orders ALTER COLUMN Placeholder VARCHAR(32)` и снова запустим запрос из листинга 7.11. Как видно на рис. 7.12, оценочный размер строки изменился с 8017 до 37 байт, отчего стало предоставляться гораздо меньше памяти.

---

[1] Автор не совсем точен. Один символ в столбце varchar может занимать разное количество байтов (от 1 до 4) в зависимости от кодировки (например, UCS-2 или UTF-8). Во многих распространенных случаях символы действительно занимают ровно по 2 байта, но это не обязательно. Параметр *n* в nvarchar означает не количество символов, а количество пар байтов. Впрочем, на смысл тезиса в книге это не влияет: оптимизатор запросов в любом случае оценивает количество байтов, а не количество символов, так что расчеты правильны. — *Примеч. ред.*

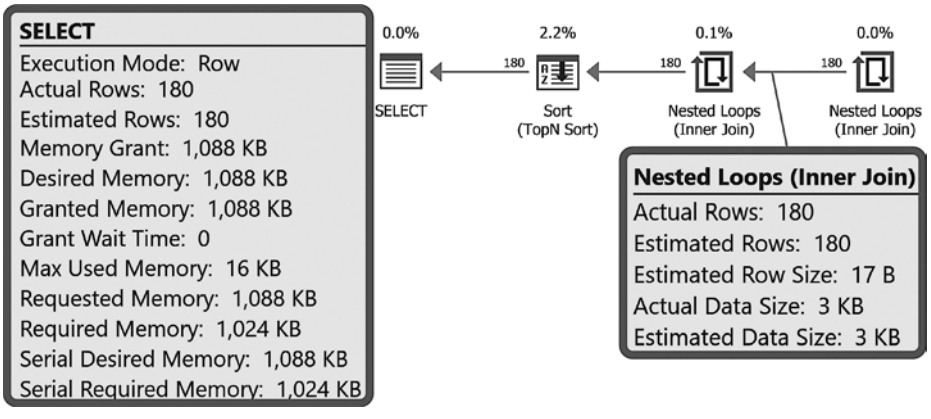


Рис. 7.12. Оптимизация запросов, интенсивно использующих память: предоставление памяти для типа VARCHAR

Обратная связь по предоставлению памяти

Функция *обратной связи по предоставлению памяти (memory grant feedback)*, появившаяся в SQL Server 2017 Enterprise Edition, позволяет SQL Server динамически корректировать предоставление памяти для кэшированных планов выполнения, опираясь на потребление памяти при предыдущих выполнениях запросов. В SQL Server 2017 обратная связь по предоставлению памяти работает только с запросами, которые выполняются в пакетном режиме, — это в основном те запросы, где участвуют индексы columnstore. В SQL Server 2019 обратная связь по предоставлению памяти стала доступна и для запросов в построчном режиме.

Эта обратная связь исправляет как избыточное, так и недостаточное предоставление памяти. При избыточном предоставлении памяти происходит перерасчет, если запрос использует менее 50 % предоставленной памяти. При недостаточном предоставлении перерасчет запускается в случае переноса в `tempdb` (подробнее об этом в главе 9). После перерасчета SQL Server обновляет параметры предоставления памяти в кэшированном плане выполнения и в дальнейшем использует новые значения.

Обратная связь в пакетном режиме доступна в базах данных с уровнем совместимости 140 (SQL Server 2017) и выше. Для обратной связи в построчном режиме нужен уровень совместимости не меньше 150 (SQL Server 2019). Она также работает в базах данных Azure SQL.

Обратную связь как в построчном, так и в пакетном режиме можно отключить на уровне базы данных командой `ALTER DATABASE SCOPED CONFIGURATION`. Обратная связь в построчном режиме регулируется параметром `ROW_MODE_MEMORY_GRANT_`

FEEDBACK. Однако обратная связь в пакетном режиме по-разному настраивается в SQL Server 2017 и 2019.

Чтобы в SQL Server 2017 отключить обратную связь в пакетном режиме, установите DISABLE_BATCH_MODE_MEMORY_GRANT_FEEDBACK в значение ON. В SQL Server 2019 установите BATCH_MODE_MEMORY_GRANT_FEEDBACK в значение OFF.

Обратную связь по предоставлению памяти можно отключить и на уровне запроса с помощью указаний DISABLE_BATCH_MODE_MEMORY_GRANT_FEEDBACK и DISABLE_ROW_MODE_MEMORY_GRANT_FEEDBACK. Это может быть полезно для планов, чувствительных к параметрам, если возникают проблемы из-за сканирования параметров. Обратная связь обычно обнаруживает такие ситуации и прекращает корректировать предоставление памяти, хотя в некоторых случаях для этого уместнее использовать указания запросов.

Наконец, в SQL Server 2019 и более ранних версиях скорректированное предоставление памяти не запоминается и применяется только к кэшированным планам. Изменения будут потеряны, если план пропадет из кэша, а также при перезапуске или аварийном переключении SQL Server. Ситуация изменилась в SQL Server 2022, где информация о предоставлении памяти остается в хранилище запросов. В этой версии внедрены и другие улучшения, которые повышают точность расчетов, связанных с предоставлением памяти.

Управление объемом предоставленной памяти

Указания запроса MIN_GRANT_PERCENT и MAX_GRANT_PERCENT позволяют задать минимальный и максимальный *процент* памяти рабочей области, который может быть предоставлен запросу. К сожалению, работать с этими указаниями сложно. SQL Server не позволяет задать объем предоставляемой памяти в абсолютных единицах, например килобайтах или мегабайтах, так что приходится указывать процент доступной памяти, которая зависит от оборудования и конфигурации SQL Server.

Память рабочей области может составлять до 75 % памяти целевого сервера. Оба этих показателя можно отслеживать с помощью счетчиков производительности Maximum Workspace Memory (KB) и Target Server Memory (KB). По умолчанию максимальный объем памяти, которую можно предоставить отдельному запросу, составляет 25 % памяти рабочей области.

Предположим, что у вас есть запрос, показанный в листинге 7.13.

Листинг 7.13. Гипотетический запрос

```
SELECT Col1, Col2
FROM T1
ORDER BY Col3
OPTION(MIN_GRANT_PERCENT=0.5,MAX_GRANT_PERCENT=3);
```

Если экземпляр SQL Server установлен на целевом сервере со 100 Гбайт памяти, то максимальный объем памяти рабочей области по умолчанию будет равен 75 Гбайт. В этой конфигурации запрос получит как минимум 0,5 % от 75 Гбайт, то есть 0,375 Гбайт (384 Мбайт), а как максимум — 3 % от 75 Гбайт, то есть 2,25 Гбайт. Стоит отметить, что этот объем может меняться в зависимости от *требуемой памяти*, необходимой для запуска запроса.

Ситуация еще более усложняется, когда задействован регулятор ресурсов, который позволяет разделить память рабочей области между несколькими пулами ресурсов, используя настройки пула ресурсов `MIN_MEMORY_PERCENT` и `MAX_MEMORY_PERCENT`. Кроме того, можно дополнительно ограничить предоставление памяти для отдельных запросов в группе рабочей нагрузки пула ресурсов, установив свойство `REQUEST_MAX_MEMORY_GRANT_PERCENT`.

Для примера предположим, что у вас есть пул ресурсов, где `MAX_MEMORY_PERCENT` установлен в 60 %, и группа рабочей нагрузки с `REQUEST_MAX_MEMORY_GRANT_PERCENT`, равным 10 %. В такой конфигурации объем предоставленной памяти для запроса из листинга 7.13 будет следующим:

- Минимальный объем: 75 Гбайт памяти рабочей области × 60 % лимита пула ресурсов × 10 % лимита группы рабочей нагрузки × 0,5 % лимита запросов = = 0,0225 Гбайт (23,04 Мбайт).
- Максимальный объем: 75 Гбайт памяти рабочей области × 60 % лимита пула ресурсов × 10 % лимита группы рабочей нагрузки × 3 % лимита запросов = = 0,135 Гбайт (138,24 Мбайт).

Хотя регулятор ресурсов и указания запросов позволяют до некоторой степени контролировать размер предоставляемой памяти, это решение довольно хрупкое. Любые изменения, влияющие на конфигурацию памяти в SQL Server, приведут к тому, что объем предоставленной памяти начнет вычисляться по-другому, а это чревато всякими неожиданностями. Используйте такие методы с особой осторожностью и только в крайнем случае, когда оптимизация запросов и обратная связь по предоставлению памяти не помогают. (Подробнее о регуляторе ресурсов можно прочитать в документации Microsoft[1].)

In-Memory OLTP и устранение неполадок

Разговор про управление памятью в SQL Server был бы неполным без упоминания технологии оперативной обработки транзакций, выполняющейся в памяти (сокращенно — In-Memory OLTP). Эта технология основана на таблицах, оптимизированных для памяти. Их данные могут для надежности

[1] https://oreil.ly/ypXV9

храниться на диске, но при запуске базы данных SQL Server целиком загружает таблицы в память.

Это сильно отличается от работы с обычными таблицами на диске, когда SQL Server загружает данные в буферный пул, но ему не приходится загружать и кэшировать целую таблицу. В памяти находится только активная часть данных из таблицы.

Что более важно, в случае нехватки памяти SQL Server может уменьшить объем буферного пула и кэшировать меньше данных. Это может повлиять на производительность системы, потому что увеличится количество физических операций ввода/вывода; однако даже в этих условиях система продолжит работать.

А в случае In-Memory OLTP SQL Server при запуске базы данных загружает в память все данные, оптимизированные для памяти. База данных не будет доступна, если на сервере недостаточно памяти для хранения данных. Более того, если по мере роста данных у SQL Server перестанет хватать памяти, то таблицы, оптимизированные для памяти, окажутся доступны только для чтения.

Если In-Memory OLTP начинает потреблять все больше и больше памяти, это может сказаться на других компонентах SQL Server, память для которых уменьшится. В Standard Edition SQL Server In-Memory OLTP может использовать не более 32 Гбайт на базу данных. У Enterprise Edition SQL Server 2016 и более поздних версий потребление памяти формально не ограничено, но на практике In-Memory OLTP занимает не более 80 % памяти выделенного регулятором пула ресурсов, к которому привязана база данных (или пула ресурсов `DEFAULT`, если привязки нет).

Это поведение можно использовать, чтобы ограничивать объем памяти, доступной для In-Memory OLTP. В листинге 7.14 показано, как это сделать. Хранимые процедуры `sys.sp_xtp_bind_db_resource_pool`[1] и `sys.sp_xtp_unbind_db_resource_pool`[2] соответственно привязывают базу данных к пулу ресурсов и отвязывают от него. Чтобы изменения вступили в силу, может понадобиться перезапустить базу данных.

Будьте осторожны и не забывайте, что при достижении предельного объема памяти данные In-Memory OLTP станут доступны только для чтения.

Листинг 7.14. Ограничение объема памяти для In-Memory OLTP

```
CREATE RESOURCE POOL InMemoryDataPool
WITH (MIN_MEMORY_PERCENT=40,MAX_MEMORY_PERCENT=40);
```

[1] https://oreil.ly/27ecr

[2] https://oreil.ly/8Uo3i

```
ALTER RESOURCE GOVERNOR RECONFIGURE;

EXEC sys.sp_xtp_bind_db_resource_pool
    @database_name = 'InMemoryOLTPDemo'
    ,@pool_name = 'InMemoryDataPool';

-- Чтобы изменения вступили в силу,
-- нужно отключить базу данных и включить ее снова
ALTER DATABASE MyDB SET OFFLINE;
ALTER DATABASE MyDB SET ONLINE;
```

Объем памяти, который потребляет In-Memory OLTP, можно отслеживать с помощью клерка памяти `MEMORYCLERK_XTP`. Если память используется интенсивно, ее потребление каждым объектом можно анализировать с помощью представления `sys.dm_db_xtp_table_memory_stats`[1]. В листинге 7.15 показан соответствующий код. Можно также использовать отчет *Memory Usage by Memory Optimized Objects* в SSMS, где предоставляются те же данные.

Листинг 7.15. Анализ потребления памяти таблицами, оптимизированными для памяти

```
SELECT
    ms.object_id
    ,s.name + '.' + t.name AS [table]
    ,ms.memory_allocated_for_table_kb
    ,ms.memory_used_by_table_kb
    ,ms.memory_allocated_for_indexes_kb
    ,ms.memory_used_by_indexes_kb
FROM
    sys.dm_db_xtp_table_memory_stats ms WITH (NOLOCK)
        LEFT OUTER JOIN sys.tables t WITH (NOLOCK) ON
            ms.object_id = t.object_id
        LEFT OUTER JOIN sys.schemas s WITH (NOLOCK) ON
            t.schema_id = s.schema_id
ORDER BY
    ms.memory_allocated_for_table_kb DESC;
```

Проанализируйте объем данных в больших таблицах, оптимизированных для памяти. В большинстве случаев не следует держать в памяти множество исторических данных; лучше распределить их между таблицами, оптимизированными для памяти, и таблицами на диске.

Обратите внимание на схемы таблиц и на столбцы `(N)VARCHAR(MAX)` и `VARBINARY(MAX)`. In-Memory OLTP работает совсем не так, как дисковые таблицы. Столбцы больших объектов (LOB) требуют значительных объемов памяти и ограничивают производительность, даже если они пустые.

[1] https://oreil.ly/Q77Ij

Еще более важно убедиться, что в системе нет длительных или неконтролируемых транзакций. В In-Memory OLTP применяется управление версиями строк. При модификации данных генерируются новые версии строк данных, которые тоже занимают память. Процесс сборки мусора в конце концов утилизирует старые версии и удаленные строки данных, но он не будет обрабатывать данные, сгенерированные после момента запуска самой старой активной транзакции. Потребление памяти будет возрастать, и в итоге это может привести к отказу системы.

В листинге 7.16 показан код, который находит 10 самых старых транзакций In-Memory OLTP. Его можно использовать, чтобы устранять неполадки, а также настраивать средства мониторинга и уведомлений.

Листинг 7.16. Обнаружение 10 самых старых транзакций In-Memory OLTP

```
SELECT TOP 10
    t.session_id
    ,t.transaction_id
    ,t.begin_tsn
    ,t.end_tsn
    ,t.state_desc
    ,t.result_desc
    ,SUBSTRING(
        qt.text
        ,er.statement_start_offset / 2 + 1
        ,(CASE er.statement_end_offset
            WHEN -1 THEN datalength(qt.text)
            ELSE er.statement_end_offset
          END – er.statement_start_offset
        ) / 2 +1
    ) AS SQL
FROM
    sys.dm_db_xtp_transactions t WITH (NOLOCK)
        LEFT OUTER JOIN sys.dm_exec_requests er WITH (NOLOCK) ON
            t.session_id = er.session_id
        CROSS APPLY sys.dm_exec_sql_text(er.sql_handle) qt
WHERE
    t.state IN (0,3) /* ACTIVE/VALIDATING */
ORDER BY
    t.begin_tsn
OPTION (RECOMPILE, MAXDOP 1);
```

In-Memory OLTP — отличная технология, которая позволяет существенно повысить пропускную способность систем OLTP. Однако это не технология типа «установил и забыл». Она требует надлежащего проектирования системы и баз данных, а также адекватного мониторинга в промышленной среде.

Чтобы больше узнать про OLTP в памяти, советую прочитать мою книгу «Expert SQL Server In-Memory OLTP, 2nd Edition» (Apress, 2017).

Резюме

SQL Server интенсивно использует память и может потреблять сотни гигабайт или даже терабайты памяти. Это совершенно нормально и позволяет повысить производительность SQL Server. Тем не менее важно правильно сконфигурировать память сервера, особенно в невыделенных средах.

Установите и настройте параметр *Maximum Server Memory*, оставив достаточно памяти для ОС и других приложений. Подумайте о том, чтобы разрешить *блокировку страниц в памяти* учетным записям SQL Server, но помните, что это может вызвать проблемы со стабильностью системы, если она настроена неправильно.

Можно также попробовать включить *выделение больших страниц* на серверах с большим объемом памяти. Однако эта функция плохо работает с индексами columnstore в версиях SQL Server до 2019 года.

Чтобы проанализировать текущее использование памяти, можно изучить, как ее потребляют различные клерки памяти. Выявите аномалии и устраните первопричины проблем.

Отслеживайте предоставление памяти и не упускайте из виду ожидания `RESOURCE_SEMAPHORE`. По возможности оптимизируйте запросы с большим объемом предоставленной памяти. Включите функцию обратной связи по предоставлению памяти, если она доступна в вашей версии SQL Server.

В следующей главе я расскажу о модели конкурентного доступа SQL Server и объясню, как устранять проблемы с блокированием и взаимными блокировками.

Чек-лист устранения неполадок

- ☐ Проверить и отрегулировать конфигурацию памяти.
- ☐ Проанализировать использование памяти с помощью представления `sys.dm_os_memory_clerks`. Устранить возможные проблемы.
- ☐ Проанализировать потребление памяти кэшем планов.
- ☐ Проанализировать потребление памяти одноразовыми нерегламентированными планами выполнения.
- ☐ Проверить наличие ожиданий `RESOURCE_SEMAPHORE`. Найти и оптимизировать запросы, потребляющие больше всего памяти.

ГЛАВА 8

Блокировки и конкурентный доступ

Блокировки (locks) в SQL Server используются, чтобы обеспечивать согласованность транзакций и требования к изоляции данных. Блокировки не позволяют обновлять одни и те же данные сразу нескольким транзакциям, а также уменьшают или даже устраняют проблемы согласованности данных, такие как «грязные», невоспроизводимые и фантомные чтения (подробнее об этом позже).

На первый взгляд модель конкурентного доступа SQL Server может показаться запутанной, но на самом деле она вполне логична и проста для понимания, если только усвоить основы. В этой главе я обзорно рассмотрю, как эта модель работает на внутреннем уровне, а также дам рекомендации по устранению блокирования (blocking) и взаимных блокировок (deadlocks)[1].

Начнем с перечня основных типов блокировок SQL Server и того, как они ведут себя на разных уровнях изоляции транзакций. Я расскажу, почему возникает блокирование и взаимные блокировки и как с этим бороться. Затем поговорим о других аспектах блокировок — в частности, об укрупнении блокировок и оптимистичном конкурентном доступе. Наконец, рассмотрим типы ожидания, связанные с блокировками, и типичные стратегии устранения неполадок.

---

[1] По историческим причинам термины *lock* и *blocking* в русскоязычной литературе переводятся похожими и иногда взаимозаменяемыми словами *блокировка* и *блокирование*, что может вызвать путаницу. В этой книге *блокировка* означает *lock* (программный объект, которым «запирается» строка или другой объект базы данных). В свою очередь, *блокирование* означает *blocking* (ситуация, когда разные сеансы или другие объекты конкурируют за установку блокировки на один и тот же объект). Слова *блокировать*, *заблокированный*, *блокирующий* и другие однокоренные (кроме, собственно, слова *блокировка*) относятся к *blocking*, а не к *lock*. — *Примеч. ред.*

Однако прежде всего — важное предупреждение: в этой главе описано, как работают блокировки в классических дисковых таблицах на основе B-деревьев. Мы *не будем* касаться обновляемых индексов columnstore, где блокировки устроены немного иначе из-за более сложной внутренней структуры. Также не пойдет речь о In-Memory OLTP, где модель конкурентного доступа работает абсолютно по-другому. Почти ничего из материала этой главы не подходит к таблицам, оптимизированным для памяти.

Устранение неполадок, связанных с блокировками в SQL Server, — это большая и сложная тема, которой можно посвятить отдельную книгу. И такая книга есть: если хотите глубже погрузиться в эту область, почитайте мою монографию «Expert SQL Server Transaction and Locking» (Apress, 2018). Здесь же ограничимся верхнеуровневым обзором.

Типы блокировок и их поведение

Как и другие СУБД общего назначения, SQL Server предназначен для работы в многопользовательских средах. Он обрабатывает различные рабочие нагрузки одновременно и поддерживает согласованность данных, когда их запрашивают и изменяют сразу несколько пользователей.

В области СУБД есть важное понятие — транзакция. *Транзакция* — это *единица работы*, которая считывает и изменяет данные, обеспечивая их согласованность и устойчивость. У транзакции есть четыре определяющих свойства: *атомарность*, *согласованность*, *изоляция* и *устойчивость* (вместе их называют *ACID* — *atomicity*, *consistency*, *isolation* и *durability*). Рассмотрим каждое из этих свойств.

Атомарность

Атомарность гарантирует, что каждая транзакция выполняется как отдельная единица работы по принципу «все или ничего»: изменения, сделанные в рамках транзакции, либо фиксируются все целиком, либо полностью откатываются.

Рассмотрим классический пример перевода денег между банковскими счетами. Эта процедура состоит из двух отдельных операций: уменьшение баланса одного счета и увеличение баланса другого. Атомарность транзакций гарантирует, что либо обе операции завершатся успехом, либо обе не состоятся, так что система не окажется в ситуации, когда общий баланс будет несогласованным.

Согласованность

Согласованность гарантирует, что каждая транзакция переводит базу данных из одного согласованного состояния в другое, не нарушая никаких ограничений, определенных базой данных.

Изоляция

Изоляция гарантирует, что изменения, внесенные в рамках транзакции, изолированы от других транзакций и невидимы для них, пока транзакция не зафиксирована. В теории изоляция транзакций должна гарантировать, что одновременное выполнение нескольких транзакций приведет систему в то же состояние, которое получилось бы, если бы эти транзакции выполнялись последовательно. Однако в большинстве реальных систем это требование часто смягчается и применяются различные уровни изоляции транзакций.

Устойчивость

Устойчивость гарантирует, что после того, как транзакция зафиксирована, все внесенные ею изменения сохранятся и не пропадут в случае сбоя системы. SQL Server обеспечивает устойчивость путем ведения журнала транзакций с опережающей записью (WAL, write-ahead logging). Транзакция не считается зафиксированной до тех пор, пока все созданные ею записи журнала не закреплены в файле журнала. Я расскажу об этом подробнее в главе 11.

Чтобы изолировать транзакции, в SQL Server используются *блокировки* (*locks*). Блокировки устанавливаются и удерживаются на *ресурсах*, таких как строки данных, страницы, разделы, таблицы (объекты) и базы данных. На внутреннем уровне блокировки представляют собой структуры в памяти, которыми управляет соответствующий компонент SQL Server — *диспетчер блокировок (Lock Manager)*.

Основные типы блокировок

Ключевой атрибут блокировки — ее тип. На внутреннем уровне SQL Server использует более 20 различных типов блокировок. К счастью, в большинстве случаев вам придется работать лишь с некоторыми из них. Рассмотрим шесть наиболее распространенных типов блокировок.

Монопольные блокировки (Exclusive, X)

Монопольные блокировки (X) используются *операциями записи*: инструкциями `INSERT`, `UPDATE`, `DELETE` и `MERGE`, которые изменяют данные. Как можно догадаться по названию, монопольность означает, что в любой момент времени только один сеанс может удерживать блокировку (X) на том или ином ресурсе.

Таким образом соблюдается самое важное правило конкурентного доступа: *несколько сеансов не могут одновременно изменять одни и те же данные*. Другие сеансы не смогут установить блокировку (X) до тех пор, пока первая транзакция не завершится и не снимет ее с измененной строки.

О монопольных блокировках стоит помнить две важные вещи. Во-первых, уровни изоляции транзакций не влияют на поведение блокировок этого типа. Их можно установить на всех уровнях изоляции, включая `READ UNCOMMITTED`.

Во-вторых, блокировки (`X`) всегда удерживаются до конца транзакции. Чем длительнее транзакция, тем дольше удерживается блокировка, что увеличивает риск блокирования (*blocking*). Как правило, чтобы снизить этот риск, стоит делать транзакции как можно короче и обновлять данные ближе к их концу, минимизируя время удержания блокировок (`X`).

Совмещаемые блокировки (Shared, S)

Совмещаемые блокировки (`S`) используются *операциями чтения*, то есть запросами `SELECT`. Опять же, как понятно из названия, несколько сеансов могут параллельно устанавливать и удерживать блокировку (`S`) на одном и том же ресурсе.

От уровня изоляции транзакций зависит, в какой момент совмещаемые блокировки устанавливаются и как долго они удерживаются. Вскоре я расскажу об этом.

Интентные блокировки: интентная монопольная (Intent Exclusive, IX), интентная совмещаемая (Intent Shared, IS) и интентная блокировка обновления (Intent Update, IU)

Обычно SQL Server устанавливает блокировки на отдельные строки данных. Это снижает вероятность того, что нескольким сеансам придется конкурировать за блокировку одних и тех же ресурсов, и улучшает конкурентный доступ. В то же время, если использовать блокировки только на уровне строк, может пострадать производительность.

Представьте ситуацию, когда сеансу требуется монопольный доступ к таблице: например, чтобы изменить ее инструкцией `ALTER`. Если бы блокировки были только на уровне строк, то сеансу пришлось бы просматривать всю таблицу, чтобы проверить наличие таких блокировок от других сеансов. Это было бы крайне неэффективно, особенно в больших таблицах.

В документации и онлайн-ресурсах часто встречаются термины «*полная блокировка*» и «*обычная блокировка*». Их используют, чтобы отличать интентные блокировки от других типов, но *технически* это разделение некорректно. На внутреннем уровне интентные блокировки ничем не отличаются от прочих.

SQL Server решает эту проблему с помощью *интентных блокировок*, которые удерживаются на уровне страницы данных или на уровне таблицы и указывают на наличие блокировок дочерних объектов. Когда сеансу нужно установить

блокировку на уровне объекта или страницы, он проверяет ее совместимость с другими блокировками (интентными или полными), которые удерживаются в таблице или на странице, вместо того чтобы просматривать таблицу (страницу) в поисках блокировок на уровне строк.

На рис. 8.1 показан пример четырех сессий, которые установили и удерживают две монопольные и две совмещаемые блокировки. Вы видите две блокировки (X) и две (S) на строках данных, две пары интентных монопольных (IX) и интентных совмещаемых (IS) блокировок на страницах данных и четыре интентные блокировки на уровне таблицы.

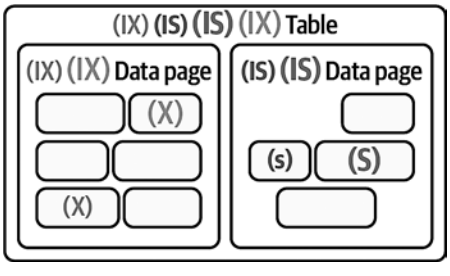


Рис. 8.1. Полная и интентная блокировки

Важно помнить еще одну вещь: *блокировки на уровне строк в SQL Server не документированы и не гарантируются*. В некоторых случаях SQL Server может использовать более крупные блокировки.

Например, когда запускается инструкция `SELECT COUNT(*) FROM T`, SQL Server должен просмотреть всю таблицу. Для этого он может использовать совмещаемые блокировки (S) на уровне страницы или даже на уровне таблицы, а не на уровне строк. Это гарантирует согласованность данных, не позволяя другим сеансам устанавливать несовместимые интентные блокировки на страницах или таблицах, а также уменьшает накладные расходы на блокировки на уровне строк.

Правила совместимости блокировок применяются всегда. Это позволяет устранять неисправности, связанные с блокировками.

Блокировки обновления (Update, U)

При модификации данных SQL Server использует блокировки обновления (U). Он устанавливает их во время *просмотра обновлений* на строки, которые нужно обновить или удалить. Установив блокировку обновления (U), SQL Server считывает строку и сверяет ее данные с предикатами запроса, чтобы понять, следует ли изменить (или удалить) ее. Если да, то SQL Server преобразует блокировку обновления в монопольную блокировку и выполняет модификацию. Если нет, он снимает блокировку обновления.

На рис. 8.2 показаны выходные данные сеанса расширенных событий, который зарегистрировал события `lock_acquired` и `lock_released` из инструкции `UPDATE`, изменившей две строки в таблице. Сначала SQL Server установил интентную эксклюзивную (`IX`) блокировку на таблице. Затем он установил интентные и «обычные» блокировки обновления (`IU`) и (`U`) на страницах и строках данных и, наконец, преобразовал их в блокировки (`IX`) и (`X`).

name	mode	resource_description	resource_type
lock_acquired	IX		OBJECT
lock_acquired	IU	3:352896	PAGE
lock_acquired	U	(8194443284a0)	KEY
lock_acquired	IX	3:352896	PAGE
lock_acquired	X	(8194443284a0)	KEY
lock_acquired	IU	3:356269	PAGE
lock_acquired	U	(0855854e442a)	KEY
lock_acquired	IX	3:356269	PAGE
lock_acquired	X	(0855854e442a)	KEY
lock_released	X	(0855854e442a)	KEY
lock_released	IX	3:356269	PAGE
lock_released	X	(8194443284a0)	KEY
lock_released	IX	3:352896	PAGE
lock_released	IX		OBJECT

Рис. 8.2. Блокировки обновления при изменении данных

Поведение блокировок обновления зависит от плана выполнения. В некоторых случаях SQL Server сначала устанавливает блокировки обновления на всех строках, а затем преобразует их в монопольные блокировки. В других ситуациях (например, если обновляется только одна строка на основе значения ключа кластеризованного индекса) SQL Server может установить монопольную блокировку, вовсе не устанавливая блокировку обновления.

Количество устанавливаемых блокировок тоже сильно зависит от плана выполнения. SQL Server не может определить, нуждается ли строка в изменении, пока не установит блокировку обновления и не прочитает строку. Бывает так, что неоптимизированная инструкция `DELETE`, которая просматривает таблицу, удаляет всего одну строку, но при этом устанавливает миллионы блокировок (`U`) в процессе просмотра обновлений, пока ищет строки, которые нужно удалить. Как вы неоднократно убедитесь в этой главе, неоптимизированные запросы сильно усугубляют проблемы конкурентного доступа.

Подобно монопольным блокировкам, блокировки обновления можно установить на всех уровнях изоляции транзакций, даже `READ UNCOMMITED`. Единственное исключение — изоляция `SNAPSHOT`, когда операции записи во время сканирования

обновлений полагаются на управление версиями строк. Я расскажу об этом позже в этой главе.

Совместимость блокировок

Пора подробнее рассмотреть совместимость блокировок. В табл. 8.1 показана матрица совместимости для основных типов блокировок. Наиболее важные правила выделены жирным шрифтом.

Таблица 8.1. Матрица совместимости блокировок

	(IS)	(IU)	(IX)	(S)	(U)	(X)
(IS)	Да	Да	Да	Да	Да	Нет
(IU)	Да	Да	Да	Да	Нет	Нет
(IX)	Да	Да	Да	Нет	Нет	Нет
(S)	Да	Да	Нет	Да	Да	Нет
(U)	Да	Нет	Нет	Да	Нет	Нет
(X)	Нет	Нет	Нет	Нет	Нет	Нет

Существуют четыре основных правила совместимости.

Интентные блокировки совместимы друг с другом

Они указывают на наличие блокировок на дочерних объектах. Несколько сеансов могут одновременно удерживать интентные блокировки на уровне объектов и страниц на одних и тех же ресурсах.

Монопольные блокировки несовместимы друг с другом и с любыми другими блокировками

Несколько сеансов не могут обновлять одни и те же данные одновременно. Более того, операции чтения, которые устанавливают совмещаемые блокировки, не могут читать незафиксированные строки, на которых удерживаются монопольные блокировки.

Блокировки обновления несовместимы друг с другом и с монопольными блокировками

Несколько операций записи не могут одновременно оценивать, нужно ли обновлять строку, а также не могут получить доступ к строке, на которой удерживается монопольная блокировка.

Блокировки обновления совместимы с совмещаемыми блокировками

Операции записи могут оценить, нужно ли обновлять строку, не блокируя операции чтения. На самом деле совместимость блокировок (`S`) и (`U`) — это основная причина, по которой SQL Server на внутреннем уровне использует блокировки обновления: они уменьшают блокирование между операциями чтения и записи.

Правила совместимости блокировок применяются всегда независимо от типа ресурса, на который устанавливаются блокировки. Помните об этом при устранении неполадок. Например, на уровне страницы часто бывает, что операция чтения, которая пытается установить блокировку (`S`), оказывается заблокирована другим сеансом, который удерживает блокировку (`IX`) на странице.

Уровни изоляции транзакций и поведение блокировок

Любая инструкция в SQL Server выполняется в составе транзакции. В SQL Server существуют три типа транзакций: явные (explicit), неявные (implicit) и автоматически фиксируемые (auto-committed).

Явные транзакции

Явные транзакции содержат несколько инструкций между явно указанными блоками `BEGIN TRAN` и `COMMIT`. Явные транзакции управляются с помощью кода, и использовать их — хорошая идея. Они помогают поддерживать согласованность данных и добавляют меньше накладных расходов, чем автоматически фиксируемые транзакции.

Неявные транзакции

Когда включены неявные транзакции, SQL Server автоматически запускает транзакцию при выполнении инструкций модификации (DML) и определения (DDL): `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `SELECT`, `CREATE` и некоторых других. Транзакция остается активной до тех пор, пока вы не завершите ее инструкцией `COMMIT` или `ROLLBACK`, аналогично явной транзакции.

Неявные транзакции редко используются в SQL Server. Чтобы их включить, можно использовать инструкции `SET IMPLICIT_TRANSACTIONS` или `SET ANSI_DEFAULTS ON`.

Автоматически фиксируемые транзакции

Когда вы запускаете инструкцию вне явной или неявной транзакции, SQL Server обрабатывает ее так, как если бы она выполнялась в собственной транзакции, то есть если бы непосредственно перед инструкцией была команда `BEGIN TRAN`, а непосредственно после инструкции — `COMMIT`. Обратите внимание, что код в триггерах выполняется в контексте той же автоматически

фиксируемой транзакции, что и основная инструкция. То же самое относится к коду хранимых процедур в инструкции `INSERT INTO .. EXEC`.

Автоматически фиксируемые транзакции, состоящие из нескольких инструкций, менее эффективны во время модификации данных, потому что добавляют накладные расходы на журнал транзакций. При этом также происходят дополнительные операции ввода/вывода, связанные с журналом (см. главу 11). По возможности лучше использовать явные транзакции, если у вас нет специальных причин этого не делать.

Когда инструкции `SELECT` запускаются вне явных или неявных транзакций, SQL Server на внутреннем уровне использует облегченные автоматически фиксируемые транзакции, которые не обращаются к журналу транзакций.

Каждая транзакция выполняется на определенном уровне изоляции транзакций, который управляет правилами согласованности данных внутри транзакции. SQL Server поддерживает шесть уровней изоляции транзакций, которые можно разделить на две категории в зависимости от того, каким образом они обеспечивают согласованность. *Пессимистичные* уровни изоляции — `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ` и `SERIALIZABLE` — работают за счет блокирования. *Оптимистичные* уровни изоляции — `READ COMMITTED SNAPSHOT (RCSI)` и `SNAPSHOT` — используют управление версиями строк.

На всех уровнях изоляции транзакций операции записи блокируют другие операции записи. Несколько сеансов не могут обновлять одни и те же данные. Кроме того, за исключением уровня `SNAPSHOT`, просмотры обновлений из нескольких сеансов блокируют друг друга. Я расскажу о поведении `SNAPSHOT` и оптимистичных уровней изоляции позже в этой главе.

Блокирование между операциями чтения и записи зависит от уровня изоляции, который контролирует *поведение и время существования совмещаемых блокировок (S) внутри транзакции, и только их*. С точки зрения конкурентного доступа от уровня изоляции не зависит поведение других типов блокировок (кроме изоляции `SNAPSHOT`, где также по-другому работают блокировки обновлений).

Инструкция `SET TRANSACTION ISOLATION LEVEL` позволяет установить уровень изоляции для транзакции (большинство клиентских библиотек SQL Server по умолчанию используют `READ COMMITTED`). Этот уровень можно переопределить на уровне таблицы в каждой отдельной инструкции с указаниями блокировки. В табл. 8.2 показано, как ведут себя совмещаемые блокировки на каждом уровне изоляции транзакции.

На уровне изоляции `READ UNCOMMITTED` совмещаемые блокировки не устанавливаются. Операциям чтения доступны строки, которые изменяются другими сеансами и на которых удерживаются монопольные блокировки. Этот уровень изоляции уменьшает блокирование, устраняя конфликты между операциями

чтения и записи, — однако это происходит за счет согласованности данных. Операции чтения считывают текущую (модифицированную) версию строки независимо от того, что произойдет дальше, даже если модификации будут отменены или если строка изменится несколько раз. Поэтому этот уровень изоляции часто называют *«грязным» чтением*.

Таблица 8.2. Уровни изоляции транзакций и поведение совмещаемых блокировок (S)

Уровень изоляции транзакции	Указание на уровне таблицы	Поведение совмещаемых блокировок
READ UNCOMMITTED	(NOLOCK) или (READUNCOMMITTED)	Не устанавливаются
READ COMMITTED (по умолчанию)	(READCOMMITTEDLOCK)	Устанавливаются и немедленно снимаются
REPEATABLE READ	(REPEATABLEREAD)	Устанавливаются и удерживаются до конца транзакции
SERIALIZABLE	(SERIALIZABLE) или (HOLDLOCK)	Совмещаемые блокировки диапазона устанавливаются и удерживаются до конца транзакции
READ COMMITTED SNAPSHOT	Отсутствует	Не устанавливаются. Используется управление версиями строк
SNAPSHOT	Отсутствует в дисковых таблицах. Указание (SNAPSHOT) поддерживается в таблицах, оптимизированных для памяти, которые выходят за рамки этой главы	Не устанавливаются. Используется управление версиями строк

Знаменитое указание NOLOCK заставляет SQL Server читать данные из таблицы в режиме READ UNCOMMITTED. Оно не влияет на операции записи: например, это указание будет проигнорировано в инструкции UPDATE WITH (NOLOCK).

На уровне изоляции READ COMMITTED SQL Server устанавливает совмещаемые блокировки перед чтением строки и снимает их сразу после чтения или, в некоторых случаях, после завершения инструкции SELECT. Это гарантирует, что транзакции не смогут прочитать незафиксированные данные из других сеансов.

На уровне изоляции REPEATABLE READ SQL Server устанавливает совмещаемые блокировки и удерживает их до конца транзакции. Это гарантирует, что другие сеансы не смогут модифицировать данные после того, как они прочитаны.

На уровне изоляции SERIALIZABLE SQL Server использует *блокировки диапазона*, удерживая их до конца транзакции. Блокировки диапазона (как монопольные,

так и совмещаемые) защищают диапазоны ключей индекса, а не отдельные строки.

Для примера рассмотрим таблицу `Orders` из двух строк со значениями `OrderId`, равными 1 и 10. На уровне изоляции `REPEATABLE READ` оператор `SELECT` установит две совмещаемые блокировки на уровне строк. Другие сеансы не смогут изменить эти строки, но смогут вставить между ними новую строку, что приведет к *фантомному чтению*: одна и та же инструкция, которая выполняется несколько раз в одной транзакции, возвращает все новые и новые добавленные строки.

На уровне изоляции `SERIALIZABLE` оператор `SELECT` устанавливает совмещаемую блокировку диапазона (`RangeS-S`), не позволяя другим сеансам вставлять данные между значениями `OrderId` 1 и 10. Это избавляет от фантомного чтения.

Оптимистичные уровни изоляции — `READ COMMITTED SNAPSHOT` и `SNAPSHOT` — не устанавливают совмещаемых блокировок. Когда операции чтения встречают строку с монопольной блокировкой, они считывают более старую (ранее зафиксированную) версию этой строки из хранилища версий в `tempdb`. Операции чтения не блокируются операциями записи и незафиксированными изменениями данных.

С точки зрения блокирования и конкурентного доступа `READ COMMITTED SNAPSHOT` ведет себя так же, как `READ UNCOMMITTED`. Оба уровня изоляции устраняют блокирование между операциями чтения и записи. Однако `READ COMMITTED SNAPSHOT` обеспечивает лучшую согласованность данных, не допуская доступ к незафиксированным данным и «грязное» чтение. В подавляющем большинстве случаев лучше использовать `READ COMMITTED SNAPSHOT`, чем `READ UNCOMMITTED`. Однако помните, что это приводит к дополнительной нагрузке на `tempdb`, за которой тоже нужно следить. Я расскажу об этом в следующей главе.

Существует несколько указаний, которые позволяют переопределить поведение блокировки. Нельзя заставить SQL Server избегать монопольных блокировок и в большинстве случаев блокировок обновления во время модификации данных. Зато можно сделать так, чтобы операции чтения и запросы `SELECT` использовали блокировки обновления и монопольные блокировки вместо совмещаемых: для этого служат указания таблиц `UPDLOCK` и `XLOCK`. Они могут пригодиться, когда нужно запретить нескольким сеансам читать одни и те же данные.

Другое указание, `READPAST`, заставляет запросы пропускать строки, на которых удерживаются несовместимые блокировки, вместо того, чтобы блокироваться. Если использовать это указание с инструкциями `UPDATE` и `DELETE`, то пропущенные строки исключаются из результатов запроса `SELECT` и не будут обновляться или удаляться.

Наконец, инструкция `SET LOCK_TIMEOUT` позволяет контролировать, как долго сеансы блокируются в ожидании, пока SQL Server установит нужную им блокировку. Если это не удается в течение назначенного периода, SQL Server принудительно завершит выполнение инструкции и вызовет исключение.

Теперь, когда вы знакомы с блокировками, давайте обсудим, как устранять неполадки блокирования.

Проблемы блокирования

Блокирование происходит, когда несколько сеансов конкурируют за один и тот же ресурс. Это состояние может быть нормальным и ожидаемым: например, несколько сеансов не могут одновременно обновлять одну и ту же строку. Но зачастую блокирование оказывается неожиданным и возникает из-за того, что запросы пытаются установить ненужные блокировки.

Решение проблем блокирования обычно состоит из трех шагов.

- Обнаружить запросы, вовлеченные в блокирование.
- Определить основную причину блокирования.
- Устранить причину.

Рассмотрим пример. Для начала создадим таблицу и заполним ее данными (листинг 8.1). Обратите внимание, что столбцы `OrderId` и `OrderNum` содержат одно и то же значение, уникальное в каждой строке.

Листинг 8.1. Создание таблицы с тестовыми данными

```
CREATE TABLE dbo.Orders
(
    OrderId INT NOT NULL,
    OrderNum VARCHAR(32) NOT NULL,
    OrderDate SMALLDATETIME NOT NULL,
    CustomerId INT NOT NULL,
    Amount MONEY NOT NULL,
    OrderStatus INT NOT NULL,
    Placeholder CHAR(400) NULL
);

;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 строки
,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 строки
,N3(C) AS (SELECT 0 FROM N2 AS T1 CROSS JOIN N2 AS T2) -- 16 строк
,N4(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N3 AS T3) -- 256 строк
,N5(C) AS (SELECT 0 FROM N4 AS T1 CROSS JOIN N4) -- 65 536 строк
,IDs(ID) AS (SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) FROM N5)
INSERT INTO dbo.Orders(OrderId,OrderNum,OrderDate,CustomerId,Amount,OrderStatus)
    SELECT
        ID,CONVERT(VARCHAR(32),ID),DATEADD(DAY,-ID % 365,GETDATE())
        ,ID % 512,ID % 100,0
    FROM
        IDs;

CREATE UNIQUE CLUSTERED INDEX IDX_Orders_OrderId
ON dbo.Orders(OrderId);
```

Теперь выполним первые два шага кода, показанного в табл. 8.3, из двух разных сеансов. Не запускайте пока инструкцию ROLLBACK, потому что нужно, чтобы условие блокирования оставалось активным.

Таблица 8.3. Создание условий блокировки

Сеанс 1 (SPID = 52)	Сеанс 2 (SPID = 53)	Комментарии
BEGIN TRAN DELETE FROM dbo.Orders WHERE OrderId = 50		Сеанс 1 устанавливает монопольную блокировку на строку с OrderId = 50
	SELECT OrderId, Amount FROM dbo.Orders WITH (READCOMMITTEDLOCK) WHERE OrderNum = '100'	Сеанс 2 заблокирован при попытке установить совмещаемую блокировку на строку с OrderId = 50
ROLLBACK		Сеанс 1 снимает монопольную блокировку. Сеанс 2 возобновляет выполнение

В SQL Server есть несколько представлений, которые помогают устранять неполадки блокирования в режиме реального времени. Давайте их рассмотрим.

Устранение неполадок блокирования в реальном времени

Ключевой инструмент для устранения неполадок блокирования в реальном времени — представление sys.dm_tran_locks[1]. В нем можно найти список активных заявок на блокировку вместе с их типом и статусом (GRANT или WAIT), информацию о ресурсах, для которых запрашиваются блокировки, и несколько других полезных атрибутов.

На рис. 8.3 показан частичный вывод представлений sys.dm_tran_locks, sys.dm_os_waiting_tasks и sys.dm_exec_requests на момент блокирования. Сеанс 53 заблокирован при попытке установить совмещаемую блокировку (S) на строку, где сеанс 52 удерживает монопольную блокировку (X). Тип ожидания LCK_M_S — это ожидание совмещаемой блокировки. Позже в этой главе я расскажу больше о типах ожидания, связанных с блокировками.

[1] https://oreil.ly/dXvU1

	resource_type	request_session_id	request_mode	request_type	request_status	resource_description	resource_associated_entity_id
1	DATABASE	52	S	LOCK	GRANT		0
2	DATABASE	53	S	LOCK	GRANT		0
3	KEY	52	X	LOCK	GRANT	(f84b73ce9e8d)	72057594068729856
4	KEY	53	S	LOCK	WAIT	(f84b73ce9e8d)	72057594068729856
5	OBJECT	52	IX	LOCK	GRANT		690101499
6	OBJECT	53	IS	LOCK	GRANT		690101499
7	PAGE	52	IX	LOCK	GRANT	3:153018	72057594068729856
8	PAGE	53	IS	LOCK	GRANT	3:153018	72057594068729856

	waiting_task_address	session_id	wait_duration_ms	wait_type	blocking_session_id	resource_description
1	0x0000020A24C7D848	53	44251	LCK_M_S	52	keylock hobtid=7205759406

	session_id	start_time	status	command	blocking_session_id	wait_type	wait_time	last_wait_type
1	53	2021-10-02 09:02:59.653	suspended	SELECT	52	LCK_M_S	292806	LCK_M_S

Рис. 8.3. Вывод динамических административных представлений во время блокирования

К сожалению, представление `sys.dm_tran_locks` само по себе не дает достаточно информации для устранения неполадок. Чтобы получить подробную картину, его нужно объединить с другими представлениями. Для этого можно запустить код из листинга 8.2 в другом сеансе.

Чтобы имена объектов разрешались корректно, код нужно запускать в контексте базы данных, вовлеченной в блокирование. Кроме того, инструкция может быть заблокирована, если запускать ее во время операций DDL, — виной тому функция `OBJECT_NAME()`, используемая в сценарии. Блокировки, установленные этой функцией, несовместимы с блокировками модификации схемы из инструкций DDL. Подробнее об этом мы поговорим позже в этой главе.

Листинг 8.2. Получение дополнительных сведений о состоянии блокирования

```
SELECT
    TL1.resource_type AS [Resource Type]
    ,DB_NAME(TL1.resource_database_id) AS [DB]
    ,CASE TL1.resource_type
        WHEN 'OBJECT' THEN
            OBJECT_NAME(TL1.resource_associated_entity_id
                ,TL1.resource_database_id)
        WHEN 'DATABASE' THEN
            'DATABASE'
        ELSE
            CASE
                WHEN TL1.resource_database_id = db_id()
                THEN
                (
                     SELECT OBJECT_NAME(object_id,TL1.resource_database_id)
                     FROM sys.partitions WITH (NOLOCK)
                     WHERE hobt_id = TL1.resource_associated_entity_id
                )
                ELSE
```

```
                    '(Run under DB context)'
            END
    END AS [Object]
    ,TL1.resource_description AS [Resource]
    ,TL1.request_session_id AS [Session]
    ,TL1.request_mode AS [Mode]
    ,TL1.request_status AS [Status]
    ,WT.wait_duration_ms AS [Wait (ms)]
    ,QueryInfo.SQL
    ,QueryInfo.query_plan
FROM
    sys.dm_tran_locks TL1 WITH (NOLOCK)
        LEFT OUTER JOIN sys.dm_os_waiting_tasks WT WITH (NOLOCK) ON
            TL1.lock_owner_address = WT.resource_address AND
            TL1.request_status = 'WAIT'
    OUTER APPLY
    (
        SELECT
            SUBSTRING(S.text, (er.statement_start_offset/2)+1,
            ((
                CASE er.statement_end_offset
                    WHEN -1 THEN DATALENGTH(S.text)
                    ELSE er.statement_end_offset
                END - er.statement_start_offset)/2)+1
            ) AS SQL
            ,TRY_CAST(qp.query_plan AS XML) AS query_plan
        FROM
            sys.dm_exec_requests er WITH (NOLOCK)
                CROSS APPLY sys.dm_exec_sql_text(er.sql_handle) S
                OUTER APPLY sys.dm_exec_text_query_plan
                (
                    er.plan_handle
                    ,er.statement_start_offset
                    ,er.statement_end_offset
                ) qp
        WHERE
            TL1.request_session_id = er.session_id
    ) QueryInfo
WHERE
    TL1.request_session_id <> @@spid
ORDER BY
    TL1.request_session_id
OPTION (RECOMPILE, MAXDOP 1);
```

На рис. 8.4 показан результат этого запроса. Его гораздо проще интерпретировать, и он содержит информацию о запущенных в данный момент пакетах и их планах выполнения. Имейте в виду, что для сеансов, в которых были удовлетворены заявки на блокировку, столбцы `SQL` и `Query Plan` содержат текущий выполняемый пакет (`NULL`, если сеанс находится в спящем режиме), а не пакет, который установил блокировку изначально.

	Resource Type	DB	Object	Resource	Session	Mode	Status	Wait (ms)	SQL	query_plan
1	DATABASE	SQLServerInternals	DATABASE		52	S	GRANT	NULL	NULL	NULL
2	OBJECT	SQLServerInternals	Orders		52	IX	GRANT	NULL	NULL	NULL
3	KEY	SQLServerInternals	Orders	(f84b73ce9e8d)	52	X	GRANT	NULL	NULL	NULL
4	PAGE	SQLServerInternals	Orders	3:153018	52	IX	GRANT	NULL	NULL	NULL
5	KEY	SQLServerInternals	Orders	(f84b73ce9e8d)	53	S	WAIT	15881	SELECT [OrderId],[Am	<ShowPlanXML
6	PAGE	SQLServerInternals	Orders	3:153018	53	IS	GRANT	NULL	SELECT [OrderId],[Am	<ShowPlanXML
7	DATABASE	SQLServerInternals	DATABASE		53	S	GRANT	NULL	SELECT [OrderId],[Am	<ShowPlanXML
8	OBJECT	SQLServerInternals	Orders		53	IS	GRANT	NULL	SELECT [OrderId],[Am	<ShowPlanXML

Рис. 8.4. Подробная информация о блокировании

В высоконагруженных системах результат кода из листинга 8.2 может оказаться довольно объемным. В сопутствующих материалах книги содержится дополнительный сценарий устранения неполадок, который выводит результаты только для заблокированных и блокирующих сеансов.

Блокирование происходит, когда два или более сеансов конкурируют за один и тот же ресурс. При устранении неполадок нужно ответить на два вопроса:

- Почему блокирующий сеанс удерживает блокировку на ресурсе?
- Почему заблокированный сеанс хочет установить блокировку на ресурс?

Оба вопроса одинаково важны, однако при анализе блокирующего сеанса можно столкнуться с проблемами. Во-первых, как я только что говорил, запрос, который отображается в данных блокирующего сеанса, не всегда соответствует инструкции, которая вызвала блокирование.

Рассмотрим ситуацию, когда сеанс выполняет несколько инструкций модификации данных в составе одной транзакции. Как вы помните, SQL Server устанавливает и удерживает монопольные блокировки в обновленных строках до конца транзакции. Каждая из этих блокировок может вызвать блокирование. К сожалению, SQL Server не отслеживает информацию об инструкциях, устанавливающих блокировки, и ее нельзя просмотреть через динамические административные представления.

Вторая проблема связана с *цепочками блокирования*, которые возникают, когда блокирующий сеанс сам блокируется другим сеансом. Обычно это происходит в высоконагруженных системах OLTP и часто связано с блокировками на уровне объекта, установленными во время изменения схемы, обслуживания индекса и некоторых других процессов.

Вот что нужно помнить о совместимости блокировок: *чтобы SQL Server удовлетворил заявку на установку блокировки, запрошенная блокировка должна быть совместима со всеми другими блокировками, запрошенными на этот же ресурс, независимо от того, были заявки на эти блокировки удовлетворены или нет*. Когда запрошенная блокировка несовместима, получаются цепочки блокирования.

Представьте ситуацию: *сеанс 1* удерживает интентную блокировку на таблице. Эта блокировка блокирует *сеанс 2*, который пытается установить полную блокировку на таблицу, например, во время перестроения индекса. Заблокированная заявка на блокировку от *сеанса 2*, в свою очередь, сама заблокирует все остальные сеансы, которые пытаются установить интентные блокировки на уровне таблицы. Не будь *сеанса 2*, другие сеансы могли бы устанавливать интентные блокировки и выполнять их параллельно с *сеансом 1*.

В такой ситуации динамические административные представления покажут массовое блокирование, и *сеанс 2* будет отображаться в качестве `blocking_session_id` у многих других сеансов. Однако первопричина блокирования в этой цепочке — *сеанс 1*, который может отображаться как `blocking_session_id` только у одного *сеанса 2*. Я рассмотрю эту ситуацию подробнее в разделе «Блокировки схемы».

Так что одного лишь количества заблокированных сеансов недостаточно, чтобы делать выводы. Просмотрите всю цепочку блокирования и найдите корневой блокировщик. (В сопутствующих материалах книги[1] есть сценарий для этого.)

Анализировать блокирующие сеансы необходимо, когда нужно решить проблему блокирования в реальном времени. Однако, чтобы найти основную причину повторяющегося блокирования, проще сперва изучить заблокированный сеанс, в котором можно увидеть заблокированную инструкцию и план ее выполнения. Проанализировав этот план, часто удается определить источник блокирования.

На рис. 8.5 показан план заблокированной инструкции (`SPID = 53` в коде в табл. 8.3).

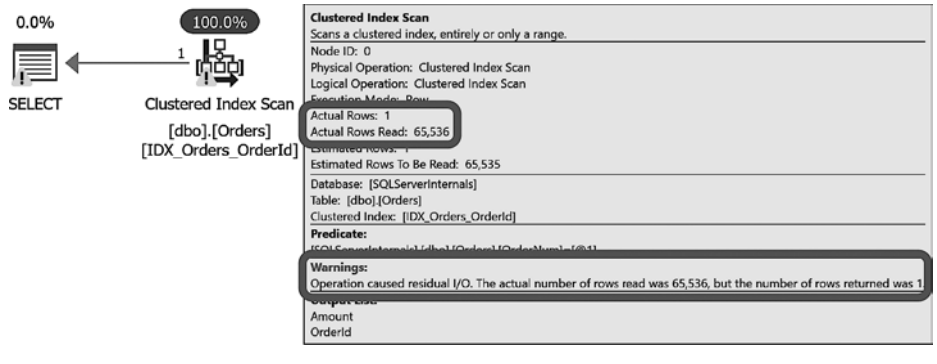


Рис. 8.5. План выполнения заблокированной инструкции

Как видите, заблокированный запрос просматривает всю таблицу в поисках заказов с предикатом столбца `OrderNum`. Запрос использует уровень изоляции транзакции `READ COMMITTED` и устанавливает совмещаемую блокировку для каж-

[1] Azure Data Studio можно скачать с сайта Microsoft: https://oreil.ly/zwwCf.

дой строки в таблице по мере чтения. Из-за несовместимости блокировок (S) и (X) в какой-то момент запрос блокируется первым оператором DELETE, который удерживает монопольную блокировку на одной из строк.

Обратите внимание, что запрос был заблокирован, несмотря на то что в строке с монопольной блокировкой не выполнялось условие `OrderNum = '100'`. Запрос SELECT не может вычислить предикат, пока не установит совмещаемую блокировку и не прочитает строку.

SQL Server должен сначала установить блокировку, чтобы понять, нужно ли читать или изменять строку. При устранении неполадок блокирования проанализируйте, какие блокировки запрос *устанавливает*, а не какие строки он возвращает или изменяет. Бывает, что запросы, которые возвращают несколько строк или даже не возвращают ни одной, устанавливают миллионы блокировок во время выполнения.

Чтобы решить эту проблему, можно оптимизировать запрос и добавить индекс в столбец `OrderNum`, отчего в плане выполнения просмотр кластеризованного индекса заменится поиском по некластеризованному индексу. При этом значительно уменьшится количество блокировок, которые устанавливает оператор, и не будет конфликта блокировок между запросами SELECT и DELETE, если только они не обрабатывают одни и те же строки.

Возможно, это слишком упрощенный пример, однако он демонстрирует самую распространенную причину блокирования: неоптимизированные запросы, которые просматривают большие объемы данных и для этого устанавливают ненужные блокировки во время просмотра. Если оптимизировать запросы, им понадобится просматривать меньше данных и устанавливать меньше блокировок, что снизит риск блокирования. На самом деле оптимизация запросов всегда уменьшает количество блокирований и взаимных блокировок.

Динамические административные представления очень полезны, чтобы устранять неполадки блокирования в режиме реального времени. К сожалению, они мало помогают, если их нельзя посмотреть в момент блокирования. Однако SQL Server помогает собирать соответствующую информацию автоматически — с помощью отчета о заблокированном процессе.

Работа с отчетом о заблокированном процессе

Отчет о заблокированном процессе (blocked process report) — это полезная функция, которая позволяет собирать информацию о состоянии блокирования для дальнейшего анализа. По умолчанию этот отчет отключен, но его можно включить, задав параметр *порогового значения заблокированного процесса (blocked process threshold)*, как показано в листинге 8.3. Этот параметр указывает, как часто SQL Server проверяет блокирования и создает отчет. Я обычно сначала

устанавливаю его равным 10–15 секунд, а потом постепенно уменьшаю так, чтобы после оптимизации он составлял 5 секунд.

Листинг 8.3. Установка 10-секундного порогового значения отчета о заблокированном процессе

```
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
EXEC sp_configure 'blocked process threshold', 10; -- в секундах
GO
RECONFIGURE;
GO
```

Когда пороговое значение установлено, отчет о заблокированном процессе можно получить с помощью расширенного события `blocked_process_report` или события трассировки SQL `Blocked process event`. Естественно, я рекомендую использовать не события трассировки, а расширенные события, которые создают меньше накладных расходов.

Отчет содержит информацию о блокирующем и заблокированном процессах в формате XML. В листинге 8.4 показана часть отчета, где выделены наиболее важные участки.

Листинг 8.4. Отчет о заблокированном процессе в формате XML

```
<blocked-process-report monitorLoop="…">
<blocked-process>
    <process id="process3e576c928" taskpriority="0" logused="0"
waitresource="KEY:
…" waittime="14102" ownerId="…" transactionname="SELECT" lasttranstarted="…"
XDES="…" lockMode="S" schedulerid="1" kpid="…" status="suspended" spid="53"
sbid="0" ecid="0" priority="0" trancount="0" lastbatchstarted="…"
lastbatchcompleted="…" lastattention="…" clientapp="…" hostname="…" hostpid="…"
loginname="…" isolationlevel="read committed (2)" xactid="…" currentdb="14"
lockTimeout="…" clientoption1="…" clientoption2="…">
        <executionStack>
            <frame line="3" stmtstart="46" sqlhandle="…"/>
            <frame line="3" stmtstart="100" sqlhandle="…"/>
        </executionStack>
<inputbuf>
SELECT OrderId, Amount
FROM dbo.Orders WITH (READCOMMITTEDLOCK)
WHERE OrderNum = '100'
        </inputbuf>
    </process>
</blocked-process>
<blocking-process>
    <process status="sleeping" spid="54" sbid="0" ecid="0" priority="0"
    trancount="1" lastbatchstarted="…" lastbatchcompleted="…" lastattention="…"
    clientapp="…" hostname="…" hostpid="…" loginname="…" isolationlevel="read
```

```
    uncommitted (1)" xactid="…" currentdb="14" lockTimeout="…" clientoption1="…"
    clientoption2="…">
        <executionStack/>
        <inputbuf>
BEGIN TRAN
    DELETE FROM dbo.Orders
    WHERE OrderId = 50
        </inputbuf>
    </process>
</blocking-process>
</blocked-process-report>
```

Как и при устранении неполадок в реальном времени, нужно изучить и блокирующий, и заблокированный процесс, чтобы найти основную причину проблемы. Для *заблокированного* процесса наиболее важна следующая информация:

`waittime`

Время, в течение которого запрос ожидает в состоянии блокирования (мс).

`lockMode`

Тип блокировки, которую запросил процесс.

`isolationlevel`

Уровень изоляции транзакций.

`executionStack` и `inputBuf`

Запрос и стек выполнения. В листинге 8.5 будет показано, как получить фактическую инструкцию SQL, участвующую в блокировании, и план ее выполнения.

У *блокирующего* процесса обратите внимание на такие свойства:

`status`

Состояние процесса: *выполняется*, *находится в спящем режиме* или *приостановлен*. Если процесс находится в спящем режиме — значит, существует незафиксированная транзакция. Если процесс приостановлен, то он либо ожидает ресурса, не связанного с блокировкой (например, страницу с диска с ожиданием `PAGEIOLATCH`), либо заблокирован другим сеансом, и тогда имеет место цепочка блокирования.

`trancount`

Любое значение больше 1 в бездействующих сеансах (в спящем режиме) или больше 2 в сеансах, выполняющих в данный момент модифицирующие инструкции (DML), указывает на вложенные транзакции. Если это происходит, когда процесс находится в спящем режиме, то существует вероятность, что клиент не зафиксировал вложенные транзакции корректно

(например, количество операторов `COMMIT` в коде меньше, чем операторов `BEGIN TRAN`).

`executionStack` и `inputBuf`

Возможно, вам придется проанализировать, что происходит внутри блокирующего процесса. К распространенным проблемам относятся неуправляемые транзакции (например, если во вложенных транзакциях нет инструкции `COMMIT`), чересчур затянувшиеся транзакции (возможно, с участием пользовательского интерфейса) и чрезмерно интенсивные просмотры (например, отсутствие индекса в ссылочном столбце подчиненной таблицы приводит к просмотрам во время проверки ссылочной целостности).

Здесь может пригодиться информация о запросах из блокирующего сеанса. Помните, что в случае блокирующего процесса `executionStack` и `inputBuf` будут соответствовать запросам, которые выполнялись в момент создания отчета о заблокированном процессе, а не в момент, когда возникло блокирование.

Как вы уже знаете, блокирование часто происходит из-за неэффективных просмотров в неоптимизированных запросах, которые устанавливают слишком много блокировок. Проверьте планы выполнения, чтобы обнаружить проблемные запросы и оптимизировать их.

Получить план выполнения можно из представления `sys.dm_exec_query_stats` на основе атрибутов `sql_handle`, `stmtStart` и `stmtEnd` из раздела стека выполнения в отчете о заблокированном процессе. Соответствующий код показан в листинге 8.5. Имейте в виду, что в плане не будут отражены фактические показатели выполнения, такие как количество выполнений и количество прочитанных строк.

Листинг 8.5. Получение плана выполнения и запроса с помощью `sql_handle`

```
DECLARE
    @H VARBINARY(MAX) = 0x00
        /* Вставить sql_handle из верхней строки стека выполнения */
    ,@S INT = 0
        /* Вставить stmtStart из верхней строки стека выполнения */
    ,@E INT = 0
        /* Вставить stmtEnd из верхней строки стека выполнения */

SELECT
    SUBSTRING(
        qt.text
        ,(qs.statement_start_offset / 2) + 1
        ,((CASE qs.statement_end_offset
            WHEN -1 THEN DATALENGTH(qt.text)
            ELSE qs.statement_end_offset
        END – qs.statement_start_offset) / 2) + 1
    ) AS SQL
```

```
    ,TRY_CAST(qp.query_plan AS XML) AS query_plan
    ,qs.creation_time
    ,qs.last_execution_time
FROM
    sys.dm_exec_query_stats qs WITH (NOLOCK)
        OUTER APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
        OUTER APPLY sys.dm_exec_text_query_plan(qs.plan_handle,@S,@E) qp
WHERE
    qs.sql_handle = @H
OPTION (RECOMPILE, MAXDOP 1);
```

К сожалению, у этого подхода есть недостаток. Как вы знаете, представление `sys.dm_exec_query_stats` опирается на кэш планов. План может быть недоступен, если в запросе используется перекомпиляция на уровне инструкций или если план почему-либо пропал из кэша. Чем дольше вы откладываете устранение неполадок, тем меньше шансов, что план будет доступен.

Эту проблему можно в некоторой степени решить, если включить хранилище запросов и изучить его данные. Кроме того, можно автоматизировать сбор данных, используя другую технологию SQL Server — уведомления о событиях (Event Notifications).

Уведомления о событиях и Blocking Monitoring Framework

В SQL Server можно получать отчеты о заблокированных процессах с помощью *уведомлений о событиях*. Эта технология основана на Брокере служб (Service Broker) и позволяет собирать информацию о конкретных событиях SQL Server и DDL и отправлять сообщения в очередь брокера. В очереди можно определить процедуру активации и реагировать на события (например, анализировать отчет о заблокированном процессе) почти в реальном времени. Это значительно повышает вероятность «поймать» планы выполнения, пока они еще есть в кэше.

Я реализовал решение на основе уведомлений о событиях, которое собирает информацию о блокировании и взаимных блокировках, разбирая данные и сохраняя их в служебной базе данных для дальнейшего анализа. (Для экономии места код здесь не приводится, но его можно найти в моем блоге и в сопутствующих материалах этой книги[1]). Я назвал этот пакет *Blocking Monitoring Framework*. На рис. 8.6 приведен пример того, как в служебной базе данных могут выглядеть собранные данные о блокировании.

Этот пакет значительно упростил мне устранение неполадок с блокировкой. Я установил его на всех промышленных серверах, которые обслуживаю, и предлагаю вам тоже попробовать его в своем окружении.

---

[1] Azure Data Studio можно скачать с сайта Microsoft: https://oreil.ly/zwwCf.

	ID	EventDate	DatabaseID	Resource	WaitTime	BlockedProcessReport	BlockedSPID	BlockedXactId
1	1	2021-10-02 10:06:10.620	5	KEY: 5:72057594068729856	1535922	<blocked-process-repo	53	241832

BlockedLockMode	BlockedIsolationLevel	BlockedSQLHandle	BlockedStmtStart	BlockedStmtEnd	BlockedQueryHash	BlockedPlanHash
S	read committed (2)	0x03000500344116	42	238	0x87D26521AC2E47	0x73D3FD64B96F8

BlockedSql	BlockedInputBuf	BlockedQueryPlan	BlockingSPID
SELECT OrderId, Amount FROM dbo.Orders WITH (READCOM	SELECT OrderId, Amount	<ShowPlanXML xmlns="http://schemas	52

BlockingStatus	BlockingTranCount	BlockingInputBuf	BlockingSql	BlockingQueryPlan
sleeping	1	BEGIN TRAN DELETE FROM dbo.Orders WHERE Or	*NULL*	*NULL*

Рис. 8.6. Информация о блокировании, собранная пакетом Blocking Monitoring Framework

Взаимные блокировки (Deadlocks)

Взаимная блокировка — это особый случай блокирования, когда несколько сеансов или несколько потоков выполнения в рамках одного сеанса блокируют друг друга. Когда это происходит, SQL Server принудительно завершает одну из транзакций, чтобы остальные могли продолжить выполнение.

Один из сеансов назначается *жертвой взаимной блокировки (deadlock victim)* — обычно тот, который создал меньше всего записей в журнале транзакций до возникновения взаимной блокировки. Этот выбор можно регулировать (до определенной степени) с помощью инструкции `SET DEADLOCK_PRIORITY`.

В классическом сценарии взаимной блокировки два или более сеансов конкурируют за один и тот же набор ресурсов. Каждый из сеансов удерживает блокировку на ресурсе, на который другой сеанс запрашивает блокировку. Таким образом сеансы блокируют друг друга, то есть создают взаимную блокировку.

Классические взаимные блокировки часто возникают в высоконагруженных системах OLTP с изменчивыми данными. Они также могут быть вызваны операциями `UPDATE`, примененными к одной строке; это связано с тем, как SQL Server обрабатывает блокировки с кластеризованными и некластеризованными индексами (об этом я расскажу чуть позже). Однако чаще всего причина взаимных блокировок — чрезмерно объемные операции просмотра и неоптимизированные запросы.

Рассмотрим пример и запустим код из табл. 8.4. Как и раньше, его нужно запустить в двух разных сеансах по одному шагу за раз.

Как вы наверняка заметили, эти сеансы работают с разными данными. Однако не забывайте о планах выполнения запросов и о том, как сеансы получают доступ к данным и устанавливают блокировки.

На первом шаге две инструкции `UPDATE` используют операцию поиска по кластеризованному индексу, чтобы обновить две отдельные строки данных и установить на них монопольные блокировки. Эти блокировки будут удерживаться до окончания транзакций.

Таблица 8.4. Создание взаимной блокировки

Сеанс 1	Сеанс 2

`BEGIN TRAN`
`  UPDATE dbo.Orders`
`  SET OrderStatus = 1`
`  WHERE OrderId = 10;`
	`BEGIN TRAN`
	`  UPDATE dbo.Orders`
	`  SET OrderStatus = 1`
	`  WHERE OrderId = 250;`

`  SELECT COUNT(*) AS [Cnt]`
`  FROM dbo.Orders WITH`
`  (READCOMMITTEDLOCK)`
`  WHERE CustomerId = 42;`
`COMMIT`
	`  SELECT COUNT(*) AS [Cnt]`
	`  FROM dbo.Orders WITH`
	`  (READCOMMITTEDLOCK)`
	`  WHERE CustomerId = 18;`
	`COMMIT`

В столбце `CustomerId` нет индекса, поэтому инструкциям `SELECT` придется выполнять просмотр кластеризованного индекса, устанавливая совмещаемые блокировки на каждую строку в таблице. В конце концов просмотр в обоих сеансах будет заблокирован из-за несовместимости блокировок (`S`)/(`X`), что приведет к взаимной блокировке. Неважно, что обновленные строки, на которых удерживается блокировка (`X`), относятся к разным клиентам и не будут включены в расчет `COUNT(*)`: SQL Server не может вычислить предикат в столбце `CustomerId`, пока не получит блокировку (`S`) и не прочитает строку.

Как вы наверняка догадались, создание некластеризованных индексов для столбца `CustomerId` решило бы эту проблему: SQL Server не придется просматривать таблицу, а сеансы не будут запрашивать несовместимые блокировки для одних и тех же строк.

Давайте посмотрим, как избавиться от взаимных блокировок.

Устранение взаимных блокировок

В целом устранение взаимных блокировок очень похоже на устранение неполадок с блокированием. Вы анализируете процессы и запросы, вовлеченные во взаимную блокировку, выявляете первопричину проблемы и, наконец, устраняете ее.

Здесь есть аналог отчета о заблокированном процессе, который называется *графом взаимной блокировки (deadlock graph)* и содержит информацию о взаимной блокировке в формате XML. Этот граф можно получить несколькими способами.

Расширенное событие `xml_deadlock_report`

Чтобы зарегистрировать событие `xml_deadlock_report`, можно создать сеанс расширенного события. Однако это событие также входит в сеанс расширен-

ного события `system_health`, который включен в SQL Server по умолчанию. Это позволяет получать информацию о графах прошлых взаимных блокировок, не настраивая какого-либо мониторинга.

В базах данных Azure SQL вместо `xml_deadlock_report` используется событие `database_xml_deadlock_report`, которое предоставляет практически такие же данные.

Событие трассировки SQL графа взаимной блокировки

Профилировщик SQL отображает графическое представление взаимной блокировки. Из него можно извлечь XML-данные графа взаимной блокировки, выбрав пункт Extract Event Data (Извлечь данные события) в контекстном меню события (щелчок правой кнопкой мыши).

Флаг трассировки `T1222`

Этот флаг трассировки сохраняет информацию о взаимных блокировках в журнале ошибок SQL Server. Этот метод можно безопасно использовать в промышленной среде, однако в наше время он избыточен, потому что существует сеанс `system_health`. Кроме того, этот флаг засоряет журнал ошибок, отчего его становится труднее анализировать.

Граф взаимной блокировки состоит из двух элементов: `<process-list>` и `<resource-list>`, которые содержат информацию о процессах и ресурсах, вовлеченных во взаимную блокировку. В листинге 8.6 показана структура графа взаимной блокировки.

Листинг 8.6. Структура графа взаимной блокировки

```
<deadlock-list>
      <deadlock victim="…">
            <process-list>
                  <process id="…">
                        …
                  </process>
                  <process id="…">
                        …
                  </process>
            </process-list>
            <resource-list>
                  <сведения о ресурсе, вовлеченном во взаимную блокировку>
                        …
                  </сведения о ресурсе, вовлеченном во взаимную блокировку>
                  <сведения о ресурсе, вовлеченном во взаимную блокировку>
                        …
                  </сведения о ресурсе, вовлеченном во взаимную блокировку>
            </resource-list>
      </deadlock>
</deadlock-list>
```

Каждый элемент `<process>` в графе содержит сведения о конкретном процессе, вовлеченном во взаимную блокировку. В листинге 8.7 показан элемент одного из процессов взаимной блокировки, которую породил код из табл. 8.4. Я удалил значения некоторых атрибутов, чтобы код было легче читать, и выделил атрибуты, которые особенно полезны при устранении неполадок.

Листинг 8.7. Элемент `<process>` графа взаимной блокировки

```
<process id="process3e4b29868" taskpriority="0" logused="264" waitresource="KEY: …"
waittime="…" ownerId="…" transactionname="… " lasttranstarted="…" XDES="…"
lockMode="S" schedulerid="…" kpid="…" status="suspended" spid="55" sbid="…"
ecid="…" priority="0" trancount="1" lastbatchstarted="…" lastbatchcompleted="…"
lastattention="…" clientapp="…" hostname="…" hostpid="…" loginname="…"
isolationlevel="read committed (2)" xactid="…" currentdb="…" lockTimeout="…"
clientoption1="…" clientoption2="…">
    <executionStack>
        <frame procname="adhoc" line="1" stmtstart="26" sqlhandle="…">
            SELECT COUNT(*) [Cnt] FROM [dbo].[Orders] with (REACOMMITTED) WHERE
[CustomerId]=@1
        </frame>
    </executionStack>
    <inputbuf>
            SELECT COUNT(*) AS [Cnt]
            FROM dbo.Orders WITH (REACOMMITTEDLOCK)
            WHERE CustomerId = 42;
        COMMIT
    </inputbuf>
</process>
```

Атрибут `id` однозначно идентифицирует процесс, а `lockMode` и `waitresource` содержат информацию о типе блокировки и ресурсе, которого ожидает процесс. В этом примере процесс ожидает совмещаемой блокировки на одной из строк (ключей). Атрибут `isolationlevel` содержит текущий уровень изоляции транзакции. Наконец, `executionStack` и `inputBuf` позволяют найти инструкцию SQL, которая выполнялась в момент возникновения взаимной блокировки.

В отличие от отчета о заблокированном процессе, `executionStack` в графе взаимной блокировки обычно предоставляет информацию о запросах и модулях, вовлеченных во взаимную блокировку. Однако в некоторых случаях может понадобиться функция `sys.dm_exec_sql_text` из кода в листинге 8.5.

Элемент `<resource-list>` графа взаимной блокировки содержит информацию о ресурсах, вовлеченных во взаимную блокировку. Пример показан в листинге 8.8.

Листинг 8.8. Элемент `<resource-list>` графа взаимной блокировки

```
<resource-list>
    <keylock hobtid="72057594039500800" dbid="14"
objectname="SqlServerInternals.dbo.Orders" indexname="PK_Orders"
        id="lock3e98b5d00"
```

```
mode="X" associatedObjectId="72057594039500800">
        <owner-list>
            <owner id="process3e6a890c8" mode="X"/>
        </owner-list>
        <waiter-list>
            <waiter id="process3e4b29868" mode="S" requestType="wait"/>
        </waiter-list>
    </keylock>
    <keylock hobtid="72057594039500800" dbid="14"
objectname="SqlServerInternals.dbo.Orders" indexname="PK_Orders"
        id="lock3e98ba500"
mode="X" associatedObjectId="72057594039500800">
        <owner-list>
            <owner id="process3e4b29868" mode="X"/>
        </owner-list>
        <waiter-list>
            <waiter id="process3e6a890c8" mode="S" requestType="wait"/>
        </waiter-list>
    </keylock>
</resource-list>
```

Имя элемента XML соответствует типу ресурса. `Keylock`, `pagelock` и `objectlock` обозначают блокировки на уровне строки, страницы и объекта соответственно. Также видно, к каким объектам и индексам относятся эти блокировки. Наконец, элементы `owner-list` и `waiter-list` описывают процессы, которые удерживают и которые ожидают блокировки соответственно, а также типы установленной и запрошенной блокировки. Эту информацию можно сопоставить с данными из элемента графа `process-list`.

Дальнейшие шаги очень похожи на действия по устранению неполадок заблокированного процесса. Вам нужно выявить запросы, вовлеченные во взаимную блокировку, и найти первопричину.

Но следует учитывать один важный фактор. В большинстве случаев во взаимную блокировку вовлечены более одной инструкции на сеанс, выполняющиеся в одной и той же транзакции. Граф взаимной блокировки содержит информацию только о *последней* инструкции — той, которая вызвала взаимную блокировку.

Следы других инструкций можно найти в элементе `resource-list`. Он показывает, что процессы удерживали монопольные блокировки на строках, но не сообщает об инструкциях, которые их установили. Чтобы изучить первопричину взаимной блокировки, полезно выявить эти инструкции, а для этого часто требуется анализировать код.

Первопричину взаимной блокировки, порожденной кодом из табл. 8.4, можно быстро обнаружить, просмотрев план выполнения инструкций `SELECT`, показанный в верхней части рис. 8.7. Здесь видно, что просмотр кластеризованного индекса привел к установке чрезмерного количества блокировок. (Чтобы получить это количество, я повторно запустил инструкцию после того, как возникла

взаимная блокировка.) Создание некластеризованного индекса для столбца `CustomerId` решит проблему, как показано в нижней части рисунка.

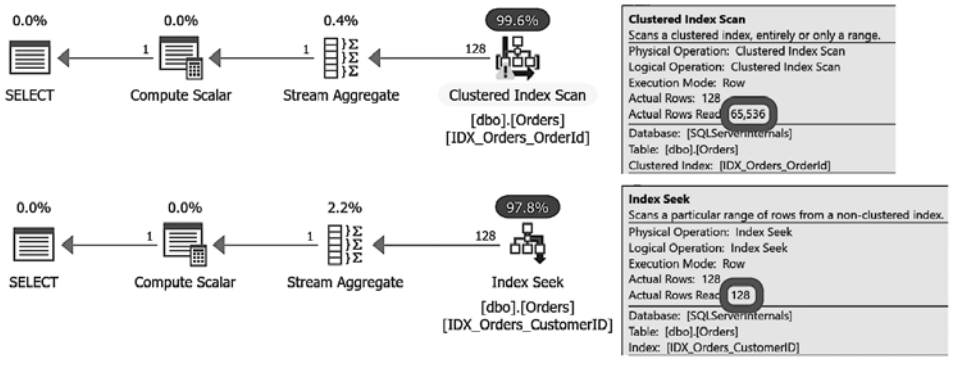


Рис. 8.7. Планы выполнения инструкции `SELECT` с некластеризованным индексом и без него

У этого подхода к устранению неполадок есть та же проблема, что и у отчета о заблокированном процессе: не всегда удается вовремя получить план выполнения запроса. К счастью, эту проблему можно решить с помощью Blocking Monitoring Framework, который собирает и анализирует информацию о взаимных блокировках в режиме реального времени.

Блокировки и индексы

Есть еще одна вещь, которую следует помнить при устранении неполадок взаимной блокировки. На физическом уровне кластеризованные и некластеризованные индексы — это отдельные объекты.

Рассмотрим таблицу, определенную в листинге 8.9.

Листинг 8.9. Определение таблицы

```
CREATE TABLE T1
(
    CI_Key INT NOT NULL,
    Col1 INT NOT NULL,
    NCI_Included_Col INT NOT NULL
);
CREATE UNIQUE CLUSTERED INDEX CI ON T1(CI_Key);
CREATE NONCLUSTERED INDEX NCI ON T1(NCI_Included_Col);
```

Когда вы обновляете эту таблицу с помощью инструкции `UPDATE`, SQL Server сначала устанавливает блокировку (`X`) на строку кластеризованного индекса. Затем он устанавливает блокировки (`X`) на строки некластеризованных индексов,

но только в тех индексах, где обновляемые столбцы являются ключевыми или неключевыми. Это очень быстрый, но не мгновенный процесс.

Когда сеанс считывает данные с помощью операций поиска по некластеризованному индексу и поиска по ключу, он устанавливает совмещаемые блокировки в обратном порядке: сначала на строки некластеризованного, а затем кластеризованного индекса. Это может привести к классической взаимной блокировке, как показано на рис. 8.8.

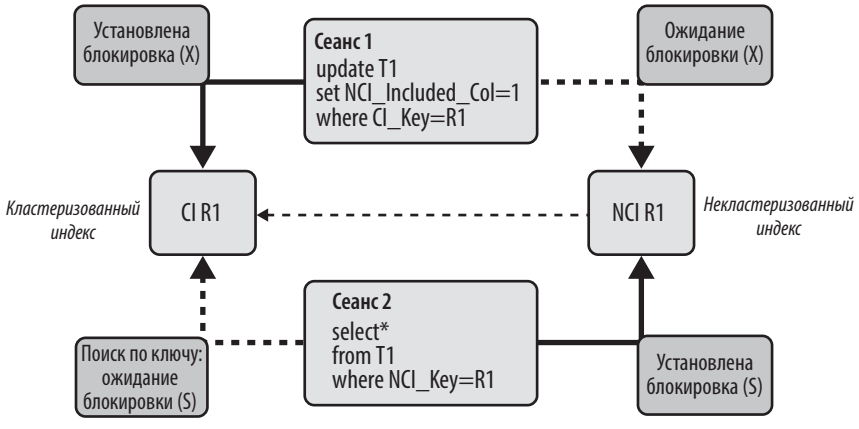


Рис. 8.8. Взаимная блокировка при поиске по ключу

Аналогичная проблема может возникнуть, когда в рамках одной транзакции вы обновляете одну и ту же строку несколько раз, изменяя столбцы, принадлежащие разным некластеризованным индексам. SQL Server не сможет установить блокировки (X) в некластеризованных индексах, пока вы не обновите столбцы в них. Это может вызвать взаимные блокировки, аналогичные той, которую мы только что рассмотрели. Чтобы решить эту проблему, можно избавиться от операции поиска по ключу с помощью покрывающих индексов или переключиться на оптимистичные уровни изоляции. Далее о них и поговорим.

Оптимистичные уровни изоляции

Оптимистичные уровни изоляции транзакций появились в SQL Server 2005 как новый способ решения проблем блокирования и конкурентного доступа, а также чтобы упростить миграцию с Oracle на SQL Server. Если на оптимистичных уровнях запросы обращаются к данным, которые изменяются другими сеансами, то запросы считывают «старые» зафиксированные версии строк, а не блокируются из-за несовместимости блокировок (S) и (X).

Существуют два оптимистичных уровня изоляции транзакций: READ COMMITTED SNAPSHOT (RCSI) и SNAPSHOT. Если говорить точнее, то SNAPSHOT — это отдельный уровень изоляции транзакций, а RCSI — это параметр базы данных, который изменяет поведение операций чтения на уровне READ COMMITTED.

Чтобы RCSI и/или SNAPSHOT работали, их нужно включить. Включить или отключить RCSI можно с помощью команды ALTER DATABASE SET READ_COMMITTED_SNAPSHOT ON/OFF. Этот параметр включен по умолчанию в базах данных Azure SQL и отключен в обычных экземплярах SQL Server. Чтобы включить или отключить RCSI, нужен монопольный доступ к базе данных, поэтому сперва придется разорвать соединения со всеми пользователями. При переключении этой опции также потребуется удалить базу данных из групп доступности AlwaysOn.

Похожим образом можно включить изоляцию SNAPSHOT с помощью команды ALTER DATABASE SET ALLOW_SNAPSHOT_ISOLATION ON. Этой инструкции не требуется монопольная блокировка, и ее можно запускать, не отключая других пользователей от базы данных.

Если включен любой из этих оптимистичных уровней изоляции транзакций, SQL Server начинает копировать «старые» версии строк в специальную часть базы данных tempdb, называемую *хранилищем версий*. Чтобы ссылаться на эти версии, в измененные строки добавляются 14-байтовые *указатели версий*. Это происходит с каждым запросом UPDATE и DELETE независимо от того, на каком уровне изоляции он выполняется. В некоторых случаях в хранилище версий может содержаться больше одной версии одной и той же строки, как показано на рис. 8.9.

В этом сценарии происходит следующее. Когда операции чтения (а иногда и записи) обращаются к строке, на которой удерживается монопольная блокировка, они считывают старую версию из хранилища версий, а не блокируются (рис. 8.10).

Оптимистичные уровни изоляции уменьшают блокирование, но добавляют накладные расходы. Прежде всего они увеличивают нагрузку на tempdb. От этого объем tempdb в изменчивых системах может значительно вырасти. В следующей главе будет рассказано, как контролировать объем и использование хранилища версий.

При модификации и извлечении данных тоже возникают накладные расходы. SQL Server должен скопировать данные в tempdb, а также поддерживать связанный список записей версий. Аналогичным образом ему требуется обходить этот список при чтении данных.

Наконец, оптимистичные уровни изоляции усугубляют фрагментацию индекса. При изменении строки SQL Server увеличивает ее размер на 14 байт из-за указателя версии. Если страница плотно упакована и новая версия строки в нее

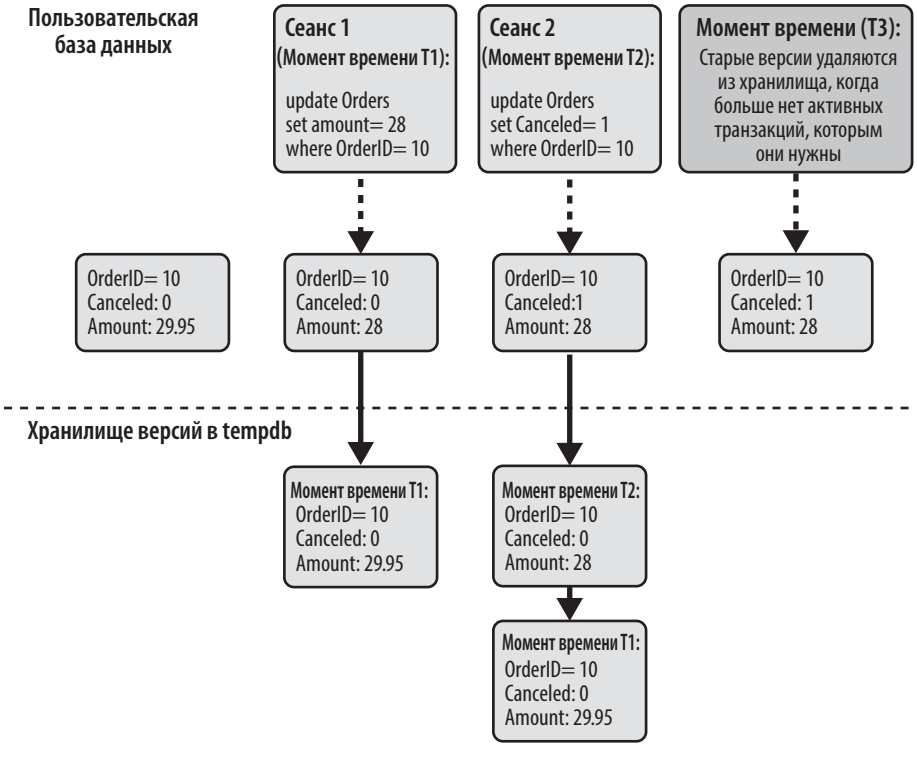


Рис. 8.9. Хранилище версий

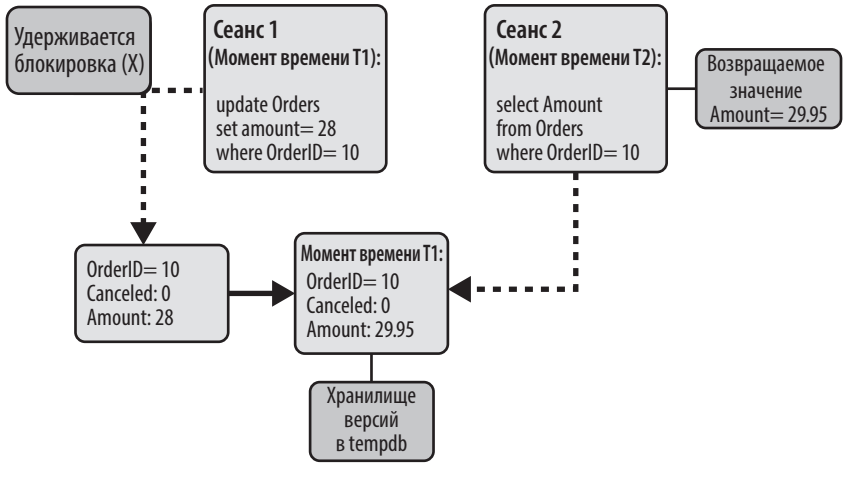


Рис. 8.10. Хранилище версий и операции чтения

не помещается, то произойдет разбиение страницы и дальнейшая фрагментация. В сопутствующих материалах книги есть любопытная демонстрация того, как инструкция DELETE может увеличить размер таблицы на диске.

Давайте обсудим оба уровня изоляции транзакций.

Уровень изоляции READ COMMITTED SNAPSHOT

Как уже говорилось, изоляция READ COMMITTED SNAPSHOT (RSCI) — это параметр базы данных, который изменяет поведение операций чтения в режиме READ COMMITTED. Когда этот параметр включен, запросы SELECT не устанавливают блокировок (S) и читают записи «старой» версии, а не блокируются из-за несовместимости блокировок (S)/(X).

Для операций записи ничего не меняется: как и на пессимистичных уровнях изоляции, запросы на изменение данных по-прежнему устанавливают блокировки (U) и (X) и могут блокировать друг друга.

На рис. 8.11 показан пример. *Сеанс 1* обновляет строку, на которой на время транзакции удерживается блокировка (X). В это время инструкция SELECT из *Сеанса 2* считывает старую версию строки из tempdb. При этом инструкции UPDATE из *Сеансов 3* и *4* блокируются из-за несовместимости блокировок (U)/(X) и (X)/(X) с блокировкой (X), удерживаемой *Сеансом 1*.

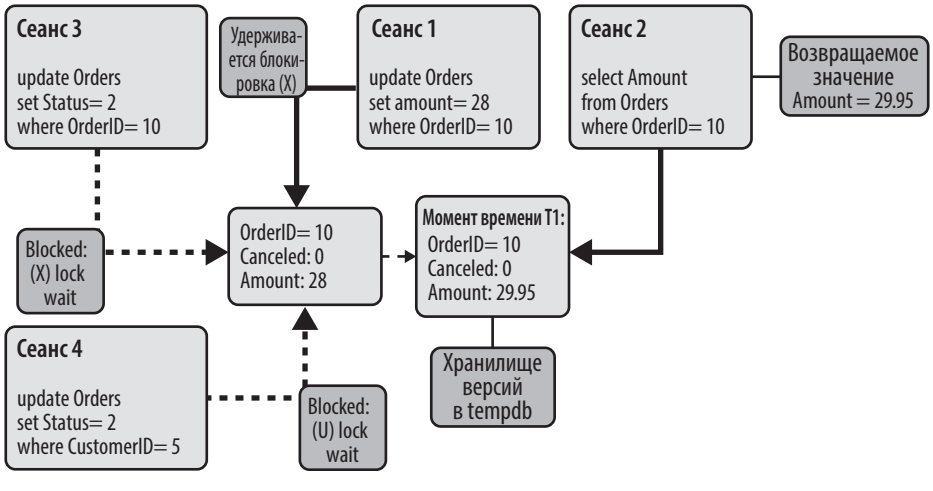


Рис. 8.11. Поведение RCSI

С точки зрения блокирования RCSI работает так же, как и READ UNCOMMITTED: оба уровня изоляции устраняют блокирование между операциями чтения и записи.

Но между этими уровнями есть большая разница: READ UNCOMMITTED устраняет блокирование за счет несогласованности данных. Возможны многие проблемы несогласованности, в том числе чтение незафиксированных данных, дублирующиеся операции чтения и пропущенные строки. В то же время RCSI обеспечивает полную *согласованность на уровне инструкций*. У инструкций на этом уровне изоляции нет доступа к незафиксированным данным и изменениям, которые были зафиксированы после запуска инструкции.

Как вы наверняка уже догадались, в запросах не следует использовать указание (NOLOCK) при включенном RCSI. Использование (NOLOCK) с READ UNCOMMITTED — это просто дурной тон, а в режиме RCSI эта подсказка вовсе бессмысленна, потому что RCSI сам по себе обеспечивает аналогичный эффект, не нарушая согласованность данных. По возможности всегда лучше использовать RCSI.

Уровень изоляции SNAPSHOT

SNAPSHOT — это отдельный уровень изоляции транзакций, не связанный с RCSI. Его нужно явно задать в коде с помощью команды SET TRANSACTION ISOLATION LEVEL SNAPSHOT.

Этот уровень изоляции обеспечивает *согласованность на уровне транзакций*. Сеансы работают со *снимком данных (data snapshot)* на момент начала транзакции. Они не видят никаких изменений данных, зафиксированных после этого момента.

Как и в случае с RCSI, не возникает блокирования между операциями чтения и записи; операции чтения не устанавливают совмещаемые блокировки. Кроме того, значительно уменьшается блокирование между операциями записи, потому что они не устанавливают блокировки обновления, а вместо этого используют записи из хранилища версий во время просмотра обновлений.

Операции записи на уровне изоляции SNAPSHOT по-прежнему устанавливают блокировки (X) при изменении данных. Они могут быть заблокированы из-за несовместимости (X)/(X), когда другая транзакция удерживает блокировку (X) на строке. Но бывает и другая ситуация: если транзакция SNAPSHOT попытается обновить строку, которая была изменена *после* запуска транзакции, то произойдет ошибка конфликта обновления и транзакция принудительно завершится. Можно обнаружить это состояние, регистрируя ошибку 3960 в блоке TRY .. CATCH, а затем при необходимости реализовать логику повторных попыток.

При переключении на уровень изоляции SNAPSHOT изменяется поведение системы. Приложения должны учитывать эти изменения и правильно их обрабатывать.

В целом оптимистичные уровни изоляции — это хороший способ борьбы с блокированием. Кроме того, включение RCSI может быть отличной экстренной мерой, которая позволяет выиграть время, если на уровне изоляции READ COMMITTED обнаруживаются существенные блокирования между операциями чтения и записи.

Но не забывайте, что временные меры могут скрыть основную причину проблем с блокированием. Уменьшение блокирования за счет оптимизации запросов требует больше времени, но в долгосрочной перспективе дает лучший результат.

Также важно помнить о накладных расходах, связанных с оптимистичными уровнями изоляции. К счастью, в большинстве случаев это не такая уж высокая плата за уменьшение проблем конкурентного доступа.

Блокировки схемы

SQL Server должен защищать метаданные базы данных, поэтому он не позволяет изменять структуру таблицы посреди выполнения запроса. Эта проблема сложнее, чем кажется. Ее не решают монопольные блокировки на уровне таблицы, потому что операции чтения на уровнях изоляции READ UNCOMMITTED, READ COMMITTED SNAPSHOT и SNAPSHOT не устанавливают блокировок (IS) на уровне таблицы.

Чтобы устранить проблему, в SQL Server используются два дополнительных типа блокировки: *модификация схемы* (Sch-M) и *стабильность схемы* (Sch-S).

Блокировки модификации схемы

Это своего рода суперблокировка. Блокировки модификации схемы (Sch-M) устанавливаются перед любыми изменениями метаданных или перед выполнением инструкции TRUNCATE TABLE. Этот тип блокировки несовместим со всеми другими типами и полностью блокирует доступ к объекту.

Подобно монопольным блокировкам, блокировки модификации схемы удерживаются до конца транзакции. Запуск инструкций DDL в явных транзакциях позволяет откатить все изменения схемы в случае ошибки, но также предотвращает любой доступ к затронутым объектам, пока транзакция не зафиксирована. Помните об этом, потому что многие инструменты сравнения схем баз данных используют явные транзакции в сценариях синхронизации схем.

Блокировки стабильности схемы

Блокировки стабильности схемы (Sch-S) применяются во время компиляции и выполнения запросов DML, когда операции чтения не используют

совмещаемые блокировки. Единственная цель блокировок стабильности — предохранять таблицу от изменения или удаления на то время, пока к ней обращается запрос. Блокировки стабильности схемы совместимы со всеми остальными типами блокировки, кроме (Sch-M).

На практике можно работать с блокировками схемы так же, как с любыми другими типами блокировок, и устранять неполадки на основе правил совместимости блокировок. Но, поскольку блокировки схемы устанавливаются на уровне объекта, эффект от блокирования может оказаться гораздо масштабнее.

Как отмечалось ранее, ключевой аспект блокировок состоит в том, что заявка на блокировку должна быть совместима со всеми другими заявками на блокировку того же ресурса независимо от того, были ли заявки уже удовлетворены или ожидают обработки в очереди. Для блокировок схемы это особенно важно.

Предположим, вы хотите изменить таблицу за пределами окна обслуживания. Оператору ALTER TABLE понадобится установить блокировку модификации схемы, которая не установится, если какие-нибудь другие сеансы удерживают блокировки на таблице. Эта заявка на блокировку (Sch-M), в свою очередь, заблокирует все другие сеансы, которые пытаются получить доступ к таблице и установить на ней интентные блокировки или блокировки стабильности схемы. Это может привести к серьезным проблемам с блокированием, как показано на рис. 8.12.

То же самое может произойти во время обслуживания индексов и разделов. И для перестроения индекса в режиме реального времени, и для переключения разделов нужно установить блокировки модификации схемы, чтобы обновить метаданные таблицы. Эти блокировки очень краткосрочны, однако они блокируют все остальные сеансы, ожидая установки.

К счастью, в обеих операциях поддерживаются *низкоприоритетные блокировки* — режим, когда блокировки схемы находятся в отдельной очереди. Заявки на блокировку ожидают там в течение определенного времени, не мешая другим сеансам устанавливать блокировки на уровне таблицы. Когда время ожидания истекает, можно либо принудительно завершить операции, либо прекратить все сеансы, удерживающие несовместимые блокировки таблиц.

Низкоприоритетные блокировки очень полезны для уменьшения блокирования при обслуживании индексов и разделов. Используйте их, когда есть такая возможность. О них также можно прочесть в документации Microsoft[1].

[1] https://oreil.ly/9p3pj

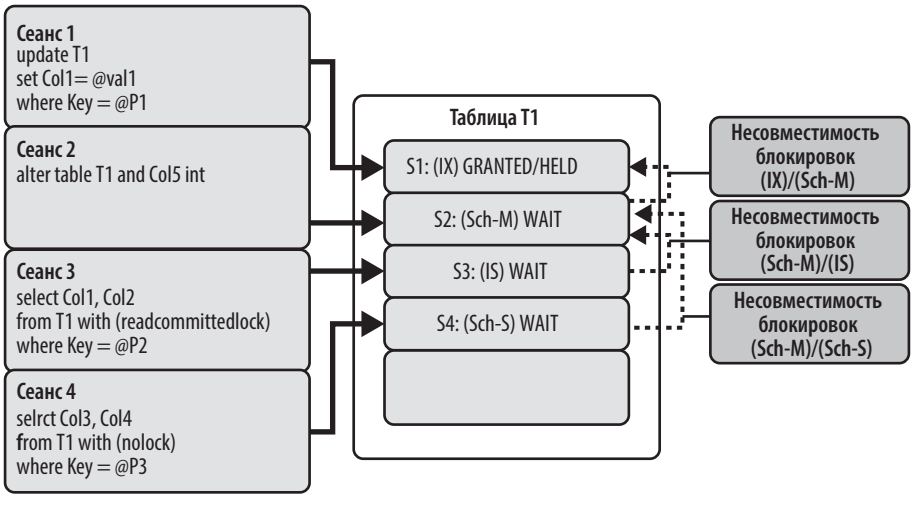


Рис. 8.12. Блокировки схемы и очередь блокировки

Укрупнение блокировок

Блокировка на уровне строк хороша с точки зрения конкурентного доступа. Но обходится она дорого. Каждая структура блокировки в 64-разрядной ОС занимает 128 байт памяти, и чтобы хранить информацию о миллионах блокировок на уровне строк и страниц, потребовались бы гигабайты оперативной памяти.

SQL Server снижает потребление памяти и затраты на управление блокировками за счет так называемого *укрупнения блокировок (lock escalation)*. Как только инструкция устанавливает *не менее* 5000 блокировок на уровне строк и страниц на один и тот же объект (индекс или кучу), SQL Server пытается заменить эти блокировки одной блокировкой на уровне таблицы или, если такая опция включена, блокировкой на уровне раздела. Эта операция завершается успешно, если никакие другие сеансы не удерживают несовместимые блокировки таблицы или раздела.

Когда укрупнение удается, SQL Server снимает все блокировки на уровне строк и страниц, удерживаемые транзакцией на объекте (или разделе), оставляя только «большую» блокировку. В случае сбоя операции SQL Server продолжает использовать блокировки на уровне строк и повторяет попытку их укрупнения каждый раз после установки примерно 1250 новых блокировок.

SQL Server также может укрупнять блокировки, когда общее количество блокировок в экземпляре превышает пороговые значения. Эти значения (5000 и 1250 блокировок) указаны приблизительно, и на практике цифры обычно

немного выше. Эти пределы также довольно малы для современных рабочих нагрузок и могут привести к непредвиденным блокированиям.

Попробуйте запустить код из листинга 8.10. Первая инструкция `ALTER TABLE` отключает укрупнение блокировок для таблицы `Orders`, о чем я расскажу позже. Не фиксируйте транзакцию, чтобы совмещаемые блокировки оставались на уровне изоляции `REPEATABLE READ`.

Листинг 8.10. Пример укрупнения блокировки (сеанс 1)

```
-- Освободить память диспетчера блокировок – не запускайте на промышленном
сервере!
-- DBCC FREESYSTEMCACHE('ALL');
GO

-- Отключить укрупнение блокировок
ALTER TABLE dbo.Orders SET (LOCK_ESCALATION = DISABLE);

-- Включить укрупнение блокировок – раскомментируйте для второй демонстрации
-- ALTER TABLE dbo.Orders SET (LOCK_ESCALATION = TABLE);
GO

DECLARE
    @C int;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
    SELECT @C = COUNT(*)
    FROM dbo.Orders WITH (ROWLOCK);

    SELECT
    (
        SELECT COUNT(*)
        FROM sys.dm_tran_locks WITH (NOLOCK)
        WHERE request_session_id = @@SPID
    ) AS [Lock Count]
    ,(
        SELECT SUM(pages_kb)
        FROM sys.dm_os_memory_clerks WITH (NOLOCK)
        WHERE [type] = 'OBJECTSTORE_LOCK_MANAGER'
    ) AS [Memory, KB]

-- Запустите код из листинга 8.11 и после этого зафиксируйте транзакцию
-- COMMIT
```

На рис. 8.13 показано количество блокировок и объем памяти, который потребляет диспетчер блокировок.

Теперь, пока транзакция еще активна, запустите код из листинга 8.11. Он вставляет в таблицу еще одну строку без какой-либо блокировки.

	Lock Count	Memory, KB
1	65652	13408

Рис. 8.13. Количество блокировок и объем потребляемой памяти без укрупнения блокировок

Листинг 8.11. Пример укрупнения блокировки (сеанс 2)

```
INSERT INTO dbo.Orders(OrderId,OrderNum,OrderDate,CustomerId,Amount,OrderStatus)
VALUES(100000,'100000',GETDATE(),1,100,0);
```

Теперь зафиксируйте транзакцию в первом сеансе (листинг 8.10) и включите укрупнение блокировки для таблицы, изменив инструкцию `ALTER TABLE` в начале сценария. Затем повторите тест (возможно, при этом понадобится изменить значение `OrderId` в листинге 8.11). Теперь совмещаемые блокировки из сеанса 1 будут укрупнены до уровня таблицы, а инструкция `INSERT` будет заблокирована из-за несовместимости блокировок (`S`)/(`IX`).

На рис. 8.14 показаны количество блокировок и объем памяти, потребленной первым сеансом, а также состояние всех заявок на блокировку на таблице (вывод из листинга 8.2).

	Lock Count	Memory, KB
1	2	1648

	Resource Type	DB	Object	Session	Mode	Status	Wait (ms)	SQL	query_plan
1	OBJECT	SQLServerInte	Orders	51	S	GRANT	NULL	NULL	NULL
2	DATABASE	SQLServerInte	DATABASE	51	S	GRANT	NULL	NULL	NULL
3	OBJECT	SQLServerInte	Orders	70	IX	WAIT	72487	INSERT INTO [dbo].[Orders	<ShowPlanXML x
4	DATABASE	SQLServerInte	DATABASE	70	S	GRANT	NULL	INSERT INTO [dbo].[Orders	<ShowPlanXML x

Рис. 8.14. Блокировки и потребление памяти в случае укрупнения блокировок

Укрупнение блокировок включено по умолчанию и может вызвать проблемы с блокированием, что иногда сбивает с толку специалистов по базам данных. Например, представьте себе процесс очистки, который удаляет большие объемы старых данных с помощью инструкции `DELETE`. Если укрупнение блокировок в сеансе очистки завершается успешно, то сеанс устанавливает и удерживает блокировку (`X`) на уровне таблицы. В результате блокируется доступ к таблице для всех операций чтения и записи на уровнях `READ COMMITTED`, `REPEATABLE READ` и `SERIALIZABLE`. Это происходит, даже если эти операции работают совсем не с тем набором данных, который очищается.

То же самое может произойти с процессом, который вставляет большой пакет строк с помощью одной инструкции `INSERT`. Как и процесс очистки, он может укрупнить монопольную блокировку (`X`) до уровня таблицы и блокировать доступ других сеансов к таблице.

У этих ситуаций есть общая черта: большое количество блокировок на уровне строк и страниц устанавливается и удерживается в рамках одной инструкции. Это вызывает укрупнение блокировок, которое удается, если никакие другие сеансы не удерживают несовместимые блокировки на уровне таблицы (или раздела).

Укрупнение блокировок запускается в зависимости от количества блокировок, установленных *отдельной инструкцией*, а не целой транзакцией. Если каждая отдельная инструкция установила меньше 5000 блокировок на уровне строк и страниц, то укрупнение не запустится.

Давайте разберемся, как обнаруживать и устранять проблемы с укрупнением блокировок.

Устранение неполадок с укрупнением блокировок

Укрупнение делает блокировку (`S`) или (`X`) на уровне таблицы видимой в выходных данных представления `sys.dm_tran_locks` и в отчете о заблокированном процессе. Примеры можно увидеть на рис. 8.14 и в листинге 8.12, где показана часть отчета о заблокированном процессе. В ней отражен заблокированный процесс, ожидающий интентной блокировки на ресурсе `OBJECT`.

Листинг 8.12. Отчет о заблокированном процессе (фрагмент)

```
<blocked-process-report>
  <blocked-process>
    <process id="…" taskpriority="0" logused="0" waitresource="OBJECT: …"
    waittime="…" ownerId="…" transactionname="user_transaction"
                             lasttranstarted="…"
    XDES="…" lockMode="IX" schedulerid="…" …>
```

Если у вас установлен Blocking Monitoring Framework, то с его помощью можно обнаруживать таблицы, в которых больше всего блокирований, связанных с интентными блокировками. (Соответствующий код входит в скачиваемый дистрибутив утилиты.)

Однако помните, что укрупнение блокировок — не единственная причина, по которой сеансы устанавливают полные блокировки объектов. Чтобы подтвердить основную причину проблемы, важно изучить отдельные случаи блокирования.

Укрупнение блокировок можно отслеживать в режиме реального времени с помощью расширенных событий. В листинге 8.13 показан код, который создает сеанс расширенного события и собирает `object_id` таблиц, где произошло укрупнение. Последняя инструкция в коде анализирует собранные данные и возвращает количество укрупнений блокировок для каждой таблицы. Обратите внимание, что вам может потребоваться изменить значение фильтра `database_id` в своей системе.

Листинг 8.13. Регистрация укрупнений блокировок с помощью расширенных событий

```
CREATE EVENT SESSION LockEscalationTracking
ON SERVER
ADD EVENT
    sqlserver.lock_escalation
    (
        WHERE database_id = 5 -- DB_ID()
    )
ADD TARGET
    package0.histogram
    (
        SET
            SLOTS = 1024 -- На основе количества таблиц в базе данных
            ,FILTERING_EVENT_NAME = 'sqlserver.lock_escalation'
            ,SOURCE_TYPE = 0 -- столбец данных о событиях
            ,SOURCE = 'object_id' -- столбец для группировки
    )
WITH
    (
        EVENT_RETENTION_MODE=ALLOW_SINGLE_EVENT_LOSS
        ,MAX_DISPATCH_LATENCY=10 SECONDS
    );

ALTER EVENT SESSION LockEscalationTracking
ON SERVER
STATE=START;
GO

-- Анализируем результаты
DECLARE
    @X XML;

SELECT @X = CONVERT(XML,st.target_data)
FROM
    sys.dm_xe_sessions s WITH (NOLOCK)
        JOIN sys.dm_xe_session_targets st WITH (NOLOCK) ON
            s.address = st.event_session_address
WHERE
    s.name = 'LockEscalationTracking' AND
    st.target_name = 'histogram';

;WITH EventInfo([count],object_id)
as
(
    SELECT
        t.e.value('@count','int')
        ,t.e.value('((./value)/text())[1]','int')
    FROM
        @X.nodes('/HistogramTarget/Slot') as t(e)
)
SELECT
    e.object_id
```

```
    ,s.name + '.' + t.name AS [table]
    ,e.[count]
FROM
    EventInfo e JOIN sys.tables t WITH (NOLOCK) ON
        e.object_id = t.object_id
    JOIN sys.schemas s WITH (NOLOCK) ON
        t.schema_id = s.schema_id
ORDER BY
    e.count desc
OPTION (RECOMPILE, MAXDOP 1);
```

Обнаружить таблицы, которые вызывают больше всего событий укрупнения блокировок, можно также с помощью столбцов `index_lock_promotion_attempt_count` и `index_lock_promotion_count` в представлении `sys.dm_db_index_operational_stats`[1] (см. главу 14).

Настройки укрупнения блокировок можно регулировать на уровне таблицы с помощью инструкции `ALTER TABLE SET LOCK_ESCALATION` (текущая настройка доступна в столбце `lock_escalation_desc` в представлении `sys.tables`). Эта инструкция влияет на укрупнение блокировок для всех индексов, определенных в таблице, — как кластеризованных, так и некластеризованных. Доступны три режима укрупнения блокировок:

`DISABLE`

Отключает укрупнение блокировок для отдельной таблицы.

`TABLE`

SQL Server укрупняет блокировки до уровня таблицы. Это режим по умолчанию.

`AUTO`

SQL Server укрупняет блокировки до уровня раздела, если таблица секционирована, или до уровня таблицы, если она не секционирована.

Два флага трассировки позволяют отключить укрупнение блокировок на уровне сеанса или экземпляра: `T1211` и `T1224`. Флаг `T1211` полностью отключает укрупнение блокировок. Флаг `T1224` запрещает укрупнение, зависящее от количества установленных блокировок, однако позволяет SQL Server укрупнять блокировки при нехватке памяти.

Эти флаги трассировки могут быть полезны, когда нужно отключить укрупнение блокировок в конкретном сеансе — например, если он импортирует или удаляет большой объем данных. Однако в последнее время я редко ими пользуюсь. В большинстве случаев я отключаю укрупнение блокировок для затронутых

[1] https://oreil.ly/Iemz7

таблиц, кроме ситуаций, когда либо на сервере очень мало памяти, либо код использует чрезвычайно большие пакетные операции, которые не удается должным образом оптимизировать.

Наконец, я хотел бы повторить, что укрупнение блокировок — это нормальное явление. Оно помогает снизить накладные расходы на управление блокировками и эффективнее использовать память, что повышает производительность системы. Держите укрупнение включенным, если только оно не вызывает заметных проблем с блокированием. Отключайте его лишь в тех таблицах, где возникают проблемы.

Чтобы оценить влияние блокирования в целом и укрупнения блокировок в частности, можно проанализировать ожидания, связанные с блокировками, в статистике ожидания. Поговорим об этом подробнее.

Ожидания, связанные с блокировками

У каждого типа блокировки есть соответствующий тип ожидания, имя которого начинается с `LCK_M_`, а заканчивается аббревиатурой типа блокировки. Например, `LCK_M_U` и `LCK_M_IX` — это ожидания блокировок обновления (`U`) и интентных монопольных блокировок (`IX`) соответственно.

Эти ожидания блокировки происходят только во время блокирования, когда сеансы приостанавливаются, пока их заявки на блокировку ожидают в очереди. SQL Server не создает ожидания блокировки, когда заявки удовлетворяются немедленно и блокирования не происходит.

Как и в случае с другими типами ожидания, смотрите как на общее время ожидания, так и на количество ожиданий. Вполне возможно, что значительное общее время ожидания создает всего несколько долгих блокирующих событий. Их можно оптимизировать или проигнорировать — в зависимости от вашей ситуации.

Ожидания, связанные с блокировками, позволяют наглядно оценить проблемы блокирования и дают направление для дальнейшего анализа.

Вероятно, сейчас вы уже понимаете, как используются различные типы блокировок, и догадываетесь, как решать те или иные проблемы конкурентного доступа. Но давайте все равно кратко пробежимся по ним.

Тип ожидания LCK_M_U

Тип `LCK_M_U` обозначает ожидание блокировок обновления (`U`), которые SQL Server устанавливает во время просмотра обновлений. Большое количество таких ожиданий обычно указывает на плохо оптимизированные запросы

записи (UPDATE, DELETE, MERGE). Часто они идут бок о бок с типами ожидания PAGEIOLATCH* и CXPACKET.

Когда я встречаю большие ожидания LCK_M_U, то обычно стараюсь оптимизировать запросы (см. главы 4 и 5). Чтобы определить целевые запросы для оптимизации, я могу установить Blocking Monitoring Framework и использовать собранные данные. Повторю, что оптимизация запросов *всегда* помогает уменьшить проблемы конкурентного доступа.

Тип ожидания LCK_M_S

Тип LCK_M_S связан с совмещаемыми блокировками (S). Эти блокировки устанавливаются запросами SELECT на уровнях изоляции READ COMMITTED, REPEATABLE READ и SERIALIZABLE.

Во многих случаях основная причина ожиданий LCK_M_S такая же, как у LCK_M_U: плохо оптимизированные запросы SELECT просматривают большие объемы данных и натыкаются на монопольные блокировки, которые удерживаются другими сеансами. Оптимизация запросов поможет решить проблему.

Когда запросы выполняются на уровне изоляции READ COMMITTED, попробуйте включить параметр базы данных READ_COMMITTED_SNAPSHOT, чтобы ликвидировать блокирование между операциями чтения и записи. Помните, что этот подход не устраняет первопричину, а лишь маскирует проблемы, вызванные плохо оптимизированными запросами. Также не забывайте о дополнительных накладных расходах, которые вносят оптимистичные уровни изоляции.

В некоторых случаях ожидания LCK_M_S могут порождаться ожиданиями блокировок на уровне таблицы, которые SQL Server устанавливает во время некоторых операций. Примером такой ситуации служит перестроение индекса в реальном времени, когда в начале операции нужна короткая блокировка (S) на уровне таблицы. Такого же эффекта можно добиться указанием (TABLOCK) в запросах.

В подобных случаях характерны ожидания, количество которых невелико, но среднее время ожидания достигает больших значений. Так или иначе, изучите отдельные эпизоды блокирования, чтобы добраться до основной причины проблемы.

Тип ожидания LCK_M_X

Тип LCK_M_X обозначает ожидание монопольных блокировок (X). Как бы странно это ни звучало, в системах OLTP с изменчивыми данными ожидания LCK_M_X возникают реже, чем ожидания LCK_M_U.

К сожалению, трудно определить наиболее распространенный сценарий, который приводит к этому типу блокирования и ожиданиям LCK_M_X. Иногда это

происходит, когда несколько сеансов обращаются к одним и тем же данным: например, в высоконагруженной реализации *таблицы счетчиков*. Другие характерные случаи — чрезмерное использование уровней изоляции `REPEATABLE READ` и `SERIALIZABLE`, неэффективное управление транзакциями и длительные транзакции, а также указания блокировки на уровне таблицы, такие как `(TABLOCKX)`.

Обнаружив большое количество ожиданий `LCK_M_X`, проанализируйте отдельные случаи блокирования, чтобы найти основную причину. Опять же, в этой ситуации чрезвычайно полезен Blocking Monitoring Framework.

Типы ожидания LCK_M_SCH_S и LCK_M_SCH_M

Типы ожидания `LCK_M_SCH_S` и `LCK_M_SCH_M` обозначают ожидания блокировок стабильности схемы (`Sch-S`) и модификации схемы (`Sch-M`). Они обычно не проявляются в больших масштабах.

Как уже отмечалось, SQL Server устанавливает блокировки (`Sch-M`) при изменении схемы. Эти блокировки требуют монопольного доступа к таблице. Пока все остальные сеансы не отключатся от таблицы, заявка на блокировку не будет удовлетворена, что приведет к ожиданию.

Есть несколько распространенных ситуаций, когда происходит такое блокирование. Во-первых, оно может случиться, если схема базы данных изменяется, когда к системе подключены другие пользователи. В таких случаях блокировка модификации схемы будет удерживаться до конца транзакции и породит ожидания блокировки стабильности схемы и интентной блокировки в других сеансах. Блокирование такого рода также может произойти во время перестроения индекса в автономном режиме, на заключительном этапе перестроения индекса в оперативном режиме или при переключении разделов. Чтобы уменьшить блокирование, можно использовать низкоприоритетные блокировки, если они поддерживаются.

Блокировки стабильности схемы применяются, чтобы избежать изменений во время использования таблицы. Они совместимы со всеми другими типами блокировок, кроме блокировок модификации схемы. Ожидания `LCK_M_SCH_S` всегда указывают на блокирование из-за модификации схемы.

Обнаружив большое количество ожиданий блокировки схемы, определите их причину. В большинстве случаев проблему можно устранить, если изменить стратегию развертывания или обслуживания базы данных или если переключиться на низкоприоритетные блокировки.

Обращайте внимание на среднее время ожидания и количество ожиданий. Типы ожидания, связанные с блокировкой схемы, обычно порождаются разовыми событиями, например неправильно запланированными развертываниями.

Типы ожидания интентной блокировки LCK_M_I*

SQL Server устанавливает интентные блокировки (`LCK_M_I*`) на уровне объекта (таблицы) и страницы. На уровне таблицы блокирование происходит в двух случаях. Первый случай — когда сеанс не может установить интентную блокировку из-за несовместимой блокировки модификации схемы. Обычно в таких ситуациях вы также увидите ожидания блокировки схемы, и с ними придется разобраться. Второй случай — когда на таблице удерживается несовместимая полная блокировка. Например, ни одна интентная блокировка не будет установлена, пока на таблице удерживается полная монопольная блокировка.

В некоторых случаях блокирование происходит из-за указаний блокировки на уровне таблицы в коде, таких как (`TABLOCK`) или (`TABLOCKX`). Однако чаще всего оно случается из-за успешного укрупнения блокировок во время больших пакетных модификаций. Когда я вижу большой процент интентных ожиданий, я первым делом анализирую и решаю проблемы с укрупнением блокировок. Обнаружить их достаточно легко, и устранение соответствующих неполадок сразу же положительно сказывается на производительности системы.

Блокировка на уровне таблицы — не единственный случай, когда возникают ожидания интентной блокировки. Я уже описывал ситуацию, когда SQL Server использует полные блокировки на уровне страницы вместо блокировок на уровне строк, чтобы просмотреть или обновить все строки на странице. Это может привести к блокированию из-за интентной блокировки, если другой сеанс пытается установить ее на страницу. Как обычно, следует выявить и устранить причину проблемы.

Типы ожидания блокировки диапазона LCK_M_R*

Помимо уровня изоляции `SERIALIZABLE`, SQL Server устанавливает блокировки диапазона (`LCK_M_R*`), сохраняя некластеризованные индексы, для которых задан параметр `IGNORE_DUP_KEY=ON`. Это может привести к проблемам конкурентного доступа и взаимным блокировкам, в которых сложно разобраться.

Обнаружив ожидания, связанные с блокировками диапазона, следует изучить отдельные случаи блокирования. К сожалению, чтобы устранить неполадки с такими ожиданиями, может потребоваться изменить стратегию транзакций или пересмотреть способ обработки проблем с качеством данных.

В этом разделе рассмотрены не все типы ожидания, связанные с блокировкой. Чтобы эффективно обрабатывать любые ожидания, важно понимать модель конкурентного доступа SQL Server. Проанализируйте, какое блокирование могло породить те или иные типы блокировок, и определите их причину.

Резюме

С помощью блокировок SQL Server защищает логическую согласованность данных во время выполнения запроса. Каждая блокировка устанавливается на тот или иной ресурс (например, на строку данных, страницу или объект) и имеет тип, определяющий совместимость и время существования блокировки.

Монопольные блокировки (`X`) устанавливаются операциями записи на строки, изменяемые инструкциями `INSERT`, `UPDATE` и `DELETE`. В каждый момент времени только один сеанс может удерживать монопольную блокировку ресурса. Монопольные блокировки устанавливаются на всех уровнях изоляции транзакций и всегда удерживаются до конца транзакции. Чтобы уменьшить блокирование, не запускайте длительные транзакции и изменяйте данные ближе к концу транзакции.

Блокировки обновления (`U`) устанавливаются операциями записи во время просмотра обновлений на всех уровнях изоляции транзакций, кроме `SNAPSHOT`. Как правило, большое количество ожиданий блокировки обновления — признак неоптимизированных запросов.

Интентные блокировки устанавливаются на уровне страницы и таблицы и указывают на наличие блокировок на дочерних объектах. Во время пакетных операций SQL Server может укрупнять блокировки на уровне строк до уровня таблиц. В таких случаях полная блокировка таблицы может мешать другим сеансам обращаться к таблице. Обнаружив заметные блокирования, связанные с интентными блокировками, устраните неполадки и отключите укрупнение блокировок для затронутых таблиц.

Совмещаемые блокировки (`S`) устанавливаются операциями чтения на уровнях изоляции `READ COMMITTED`, `REPEATABLE READ` и `SERIALIZABLE`. Единственный аспект конкурентного доступа , который зависит от уровня изоляции транзакций, — это поведение совмещаемых блокировок (за исключением блокировок обновления в режиме `SNAPSHOT`). Большое количество ожиданий совмещаемых блокировок часто связано с плохо оптимизированными запросами.

При устранении неполадок, связанных с конкурентным доступом, важно понимать, почему сеансы установили блокировки, которые привели к блокированию и взаимным блокировкам. Одна из самых распространенных причин — неоптимизированные запросы, которые просматривают большие объемы данных. Я не устаю повторять: оптимизация запросов всегда идет на пользу.

В следующей главе мы рассмотрим базу данных `tempdb`, ее производительность и потенциальные проблемы.

Чек-лист устранения неполадок

- ☐ Собрать информацию, связанную с блокированием и взаимными блокировками.
- ☐ Просмотреть типы ожидания блокировки в статистике ожидания.
- ☐ Проанализировать укрупнение блокировок, если обнаружено много ожиданий интентных блокировок.
- ☐ Пересмотреть стратегии обслуживания индексов и разделов, а также методы развертывания, если обнаружено много ожиданий блокировки схемы.
- ☐ Оптимизировать запросы, если обнаружено много ожиданий блокировок обновления и совмещаемых блокировок.
- ☐ Попробовать включить `RCSI` и использовать оптимистичные уровни изоляции, если на уровне `READ COMMITTED` обнаружены ожидания совмещаемой блокировки.
- ☐ Изучить отдельные случаи блокирования и взаимной блокировки. Обработать самые распространенные из них.
- ☐ В любом случае оптимизировать запросы: обычно это улучшает конкурентный доступ.

ГЛАВА 9

Работа с базой данных tempdb и ее производительность

Tempdb — это системная база данных, которую совместно используют все пользовательские и системные сеансы. В ней хранятся пользовательские и встроенные временные объекты и данные, к ней обращаются многие процессы. Высокая эффективность и пропускная способность tempdb крайне важны для производительности сервера.

Эту главу я начну с обзора потребителей tempdb и типичных сценариев использования, а также рассмотрю несколько профессиональных приемов работы с временными объектами. Далее я покажу, как диагностировать и устранять распространенные проблемы с tempdb. Наконец, я дам несколько советов по конфигурации tempdb.

Временные объекты в tempdb

Настройка производительности tempdb — сложная тема. Обычно я предпочитаю начинать с характерных сценариев ее использования. В конце концов, tempdb — это просто еще одна база данных, и если снизить нагрузку на нее, то производительность обычно повышается. У tempdb есть некоторые внутренние механизмы оптимизации, о которых я расскажу в этой главе, но с практической точки зрения ее можно считать такой же базой данных, как прочие.

В базе данных tempdb размещаются временные объекты, созданные пользователями, внутренние наборы записей, сформированные при выполнении запросов, хранилище версий и несколько других объектов. Запустив код из листинга 9.1, можно посмотреть, сколько места занимают различные типы объектов.

Листинг 9.1. Пространство, занимаемое объектами в базе tempdb

```
SELECT
    CONVERT(DECIMAL(12,3),
        SUM(user_object_reserved_page_count) / 128.
```

```
    ) AS [User Objects (MB)]
    ,CONVERT(DECIMAL(12,3),
        SUM(internal_object_reserved_page_count) / 128.
    ) AS [Internal Objects (MB)]
    ,CONVERT(DECIMAL(12,3),
        SUM(version_store_reserved_page_count) / 128.
    ) AS [Version Store (MB)]
    ,CONVERT(DECIMAL(12,3),
        SUM(unallocated_extent_page_count) / 128.
    ) AS [Free Space (MB)]
FROM
    tempdb.sys.dm_db_file_space_usage WITH (NOLOCK);
```

Начнем обсуждение с первой группы из этого списка: объекты, созданные пользователями.

Временные таблицы и табличные переменные

Временные таблицы и табличные переменные хранят информацию с коротким сроком существования и промежуточные результаты обработки данных. По большей части временные таблицы ведут себя так же, как обычные пользовательские таблицы. Они не поддерживают триггеры и не могут входить в представления, зато поддерживают индексы и ограничения. Их можно изменять инструкцией `ALTER`, как и обычные таблицы, однако делать этого не стоит, и позже я объясню почему.

Существуют два типа временных таблиц: глобальные и локальные. Они различаются временем существования и видимостью. Имена *глобальных временных таблиц* начинаются с двух решеток (`##`), и эти таблицы видны всем сеансам. Они удаляются, когда отключается сеанс, в котором они были созданы, если больше нет сеансов, ссылающихся на них.

Глобальные временные таблицы полезны для хранения временных данных, чтобы разные сеансы могли их совместно использовать. Но такой подход хрупок и подвержен ошибкам. Например, если сеанс, создавший глобальную временную таблицу, теряет соединение с базой данных, то таблица может исчезнуть в непредсказуемый момент. Вместо временных таблиц лучше создавать обычные таблицы в `tempdb`.

Глобальные временные таблицы можно инициализировать в хранимых процедурах запуска. Эти таблицы останутся активными и не пропадут, пока работает SQL Server.

Имена *локальных временных таблиц* начинаются с одной решетки (`#`), и эти таблицы видны только в том сеансе, в котором были созданы. Если несколько

сеансов параллельно создают локальные временные таблицы с одним и тем же именем, то у каждого сеанса будет свой отдельный экземпляр таблицы.

Локальные временные таблицы видны в модуле, в котором они были созданы, и во всех других модулях, вызываемых из него. Например, если вы открываете соединение и в этом сеансе создаете временную таблицу, то она будет видна всюду в этом сеансе и останется активна, пока сеанс открыт. Аналогично если создать временную таблицу в хранимой процедуре, то она будет видна в этой хранимой процедуре и во всех вызываемых из нее модулях T-SQL или сценариях динамического SQL. Таблица будет удалена автоматически, когда хранимая процедура завершится.

Это поведение можно использовать, чтобы передавать данные между модулями T-SQL. Но у него есть недостатки. Во-первых, при этом в SQL Server 2017 и более ранних версиях увеличиваются количество компиляций и загрузка процессора. Приходится перекомпилировать внутренний (вызванный) модуль, потому что SQL Server ничего не узнает о внешней таблице, пока модуль не будет вызван. В SQL Server 2019 эту проблему выправили: там повторная компиляция не выполняется, пока структура временной таблицы остается неизменной.

Во-вторых, этот подход чрезвычайно хрупок. Если внешние (вызывающие) модули изменяют схему временной таблицы, то внутренние модули могут сломаться. Еще хуже, если внутренние модули вызываются из нескольких внешних объектов, потому что такую структуру будет очень трудно поддерживать. Используйте этот подход с особой осторожностью и только при крайней необходимости.

Наоборот, табличные переменные видны только в том модуле, где они были определены. Их можно передавать в качестве параметров другим модулям (подробнее об этом позже в этой главе)[1].

Вопреки расхожему мифу обычные табличные переменные не являются объектами в памяти. (Типы таблиц, оптимизированные для памяти, позволяют создавать табличные переменные в памяти, но в этой главе мы их не рассматриваем.) Они, как и временные таблицы, хранятся в `tempdb`. При этом они создают меньше накладных расходов, чем временные таблицы, но с важным ограничением: табличные переменные не поддерживают статистику индекса. Это может привести к значительным ошибкам при оценке количества элементов и крайне неэффективным планам выполнения.

Рассмотрим код в листинге 9.2. Здесь я создаю временную таблицу и заполняю ее данными.

---

[1] В статье Эрланда Соммарскога (https://oreil.ly/mOOrt) описано несколько методов передачи данных между модулями T-SQL.

Листинг 9.2. Оценки количества элементов: создание временной таблицы

```
CREATE TABLE #TT(ID INT NOT NULL PRIMARY KEY);

;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 строки
,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 строки
,N3(C) AS (SELECT 0 FROM N2 AS T1 CROSS JOIN N2 AS T2) -- 16 строк
,N4(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N3 AS T2) -- 256 строк
,IDs(ID) AS (SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) FROM N4)
INSERT INTO #TT(ID)
    SELECT ID FROM IDs;
```

Теперь запустите код из листинга 9.3. Он выбирает данные из временной таблицы и табличной переменной и сравнивает оценочное количество элементов в запросах.

(Обратите внимание, что я запускаю эту демонстрацию в SQL Server 2017 в базе данных с уровнем совместимости 140. Код будет вести себя по-другому на уровне совместимости 150 или выше, как в SQL Server 2019 и более поздних версиях. Вскоре я расскажу об этом.)

Листинг 9.3. Оценки количества элементов: выбор данных из временных объектов

```
DECLARE
    @TTV TABLE(ID INT NOT NULL PRIMARY KEY);

INSERT INTO @TTV(ID)
    SELECT ID FROM #TT;

SELECT COUNT(*) FROM #TT;
SELECT COUNT(*) FROM @TTV;
SELECT COUNT(*) FROM @TTV OPTION (RECOMPILE);
```

На рис. 9.1 показаны оценки количества элементов для запросов `SELECT`. Как видите, оценка для временной таблицы верна. Однако если инструкция не перекомпилируется, то SQL Server считает, будто у табличной переменной только одна строка. (Инструкция может перекомпилироваться либо из-за параметра `OPTION (RECOMPILE)`, либо из-за других факторов.) Ошибочные оценки количества элементов могут быстро распространяться по плану выполнения, а это значит, что при использовании табличных переменных могут получиться крайне неэффективные планы.

Перекомпиляция инструкции с параметром `OPTION (RECOMPILE)` предоставляет оптимизатору запросов информацию об общем количестве строк, однако табличные переменные не хранят ни статистику, ни сведения о распределении данных.

Давайте повторим тест, добавив к запросам предложение `WHERE`, как показано в листинге 9.4. У всех строк в таблицах заданы положительные значения `ID`.

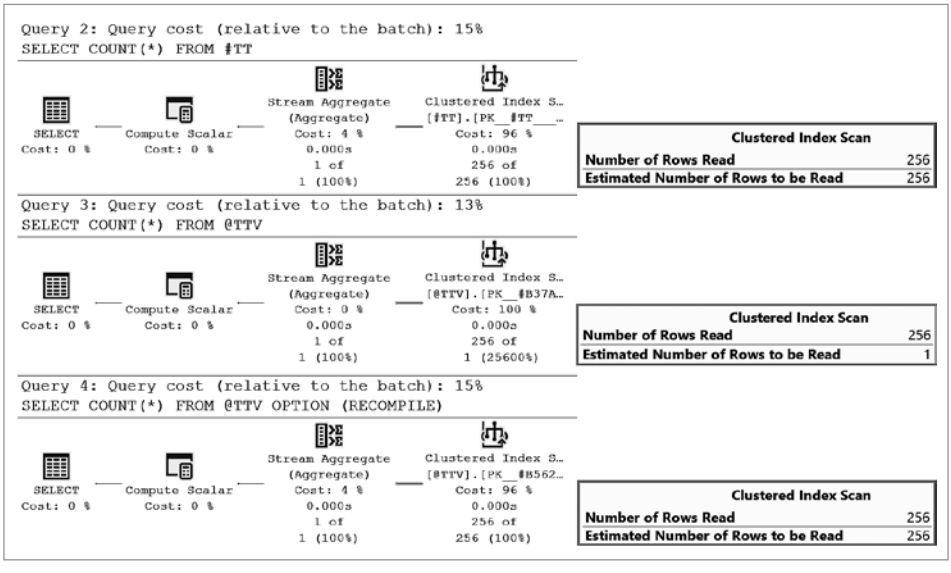


Рис. 9.1. Оценки количества элементов: временные таблицы и табличные переменные

Листинг 9.4. Оценки количества элементов: выбор данных с помощью WHERE

```
DECLARE
    @TTV TABLE(ID INT NOT NULL PRIMARY KEY);

INSERT INTO @TTV(ID)
    SELECT ID FROM #TT;

SELECT COUNT(*) FROM #TT WHERE ID > 0;
SELECT COUNT(*) FROM @TTV WHERE ID > 0;
SELECT COUNT(*) FROM @TTV WHERE ID > 0 OPTION (RECOMPILE);
```

На рис. 9.2 показаны оценки количества элементов для сценария из листинга 9.4. Временные таблицы хранят статистику по индексам, поэтому SQL Server смог правильно оценить количество строк в первой инструкции SELECT.

Как и прежде, без перекомпиляции на уровне инструкций SQL Server предполагает, будто у табличной переменной только одна строка. Но даже с перекомпиляцией оценки далеки от истины. Поскольку нет статистики, SQL Server предполагает, что оператор «больше» (>) отфильтрует около 70 % строк из таблицы, что в корне неверно.

Если запускать эти сценарии в базе данных с уровнем совместимости 150 или выше (SQL Server 2019), то во втором запросе получатся разные оценки. SQL Server откладывает компиляцию инструкций с табличными переменными и использует количество строк, имеющееся на момент компиляции, а потом

кэширует сгенерированный план. Но статистики по-прежнему нет, и если используется предложение WHERE, то оценка окажется неверной. Запомните это поведение.

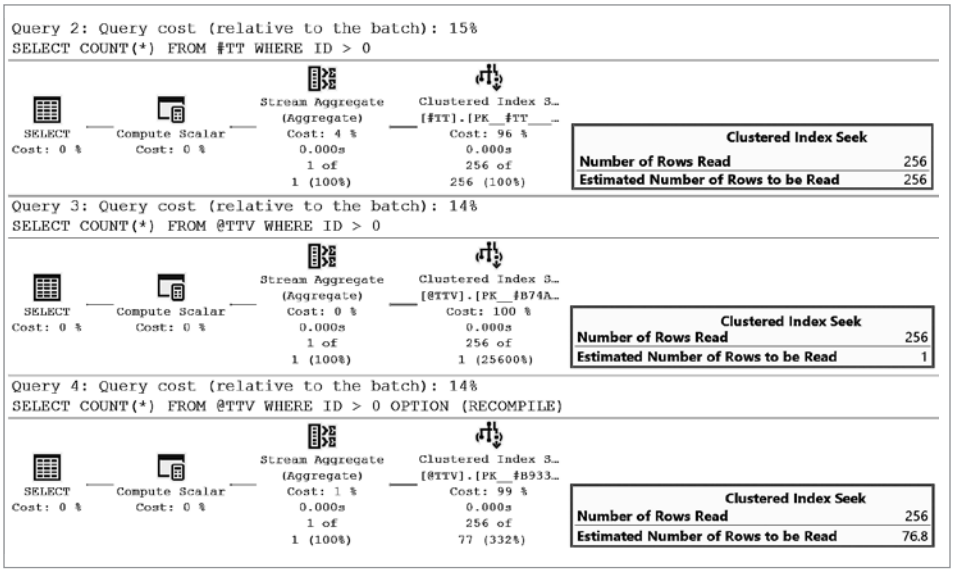


Рис. 9.2. Оценки количества элементов: табличные переменные и предложение WHERE

Не стоит использовать табличные переменные, когда ошибочные оценки количества элементов могут повлиять на планы выполнения: например, если табличные переменные хранят большие объемы данных и участвуют в соединениях с другими таблицами. В большинстве случаев оптимизатор запросов выберет соединение в цикле, что неэффективно при больших объемах входных данных. Вместо этого используйте временные таблицы: они *гораздо* безопаснее, чем табличные переменные. Зачастую производительность запросов значительно повышается, если просто заменить табличные переменные на временные таблицы.

Не забывайте правильно индексировать временные таблицы, с которыми работаете. Все принципы оптимизации, которые вы узнали в главе 5, применяются для временных таблиц так же, как для обычных.

Временные таблицы отлично помогают улучшить оценку количества элементов и оптимизировать сложные запросы. Сложный запрос можно разбить на несколько более простых и сохранить промежуточные результаты во временной таблице. Небольшие запросы легче оптимизировать, особенно с учетом того, что во временных таблицах поддерживается актуальная статистика.

Как обычно, у этого подхода есть и обратная сторона: накладные расходы на создание и заполнение временных таблиц и табличных переменных. На рис. 9.3 приведен общий обзор этого процесса.

Рис. 9.3. Использование временной таблицы для хранения промежуточных данных

Разберем ход этого процесса подробнее.

Шаг 1: Создание объекта

Когда SQL Server создает временную таблицу или табличную переменную, он вносит несколько изменений в страницы системных каталогов и карт размещения в `tempdb`. Это очень быстрый процесс, но в нем возможны состязания, когда несколько сеансов одновременно обновляют системные страницы.

Если грамотно настроить `tempdb`, то эту проблему можно уменьшить, о чем я расскажу позже в этой главе. Я также объясню, как ее правильно диагностировать, — в этой главе и главе 10.

Шаг 2: Заполнение временного объекта

Чтобы получить данные, которыми будет заполнен временный объект, SQL Server выполняет логическое или физическое чтение из исходных таблиц.

Шаг 3: Выделение страниц

SQL Server выделяет страницы данных для временного объекта в буферном пуле и модифицирует их, помечая как «грязные».

Шаг 4: Запись журналов

SQL Server регистрирует предыдущие действия в журнале транзакций базы `tempdb`. Протоколирование в `tempdb` эффективнее, чем в пользовательских базах данных, но все равно возникают дополнительные расходы, когда `tempdb` заполняется временными объектами.

Также стоит упомянуть, что табличные переменные журналируются эффективнее, чем временные таблицы. Однако это преимущество обычно сводится на нет из-за ошибочной оценки количества элементов.

Шаг 5: Запись страниц данных

В отдельном асинхронном процессе «грязные» страницы данных в конечном итоге будут записаны в файлы данных. Это может произойти даже после того, как объект удален из базы данных.

Страницы данных, принадлежащие временным объектам, остаются в буферном пуле и ведут себя как страницы обычных таблиц. Они занимают место в буферном пуле и тем самым уменьшают объем памяти, доступной для кэширования данных из обычных таблиц. Чтобы анализировать, как `tempdb` использует буферный пул, можно использовать код из листинга 7.4.

Шаг 6: Удаление объекта

В конце концов временные объекты удаляются. Хотя эта операция относительно несложная, она также изменяет системные каталоги и может привести к состязаниям в высоконагруженных системах.

Как видите, создавать и заполнять временные объекты может оказаться ресурсоемкой операцией, особенно при работе с большими объемами данных. Временные таблицы — это отличный инструмент оптимизации запросов для небольших промежуточных результатов, но вряд ли в них стоит хранить миллионы строк.

Более того, чтение данных обходится значительно дешевле, чем запись. Может оказаться быстрее и экономичнее несколько раз прочитать большие объемы данных из обычной таблицы, чем записывать крупные фрагменты оттуда во временную таблицу.

Не существует однозначных пороговых значений, указывающих, когда можно или нельзя использовать временные таблицы. Все зависит от вашей рабочей нагрузки, объема данных, конфигурации оборудования и задач, которые вы пытаетесь решить. Используя временные таблицы, убеждайтесь, что связанные с ними накладные расходы не перевешивают преимущества.

Кэширование временных объектов

Я только что упомянул, что для создания и удаления временных объектов SQL Server вносит изменения в страницы системных каталогов и карт размещения в `tempdb`. Уменьшить связанные с этим накладные расходы помогает *кэширование временных объектов* — один из механизмов оптимизации SQL Server.

Само название этого механизма немного сбивает с толку. Оно относится к кэшированию страниц *размещения* временных объектов, а не страниц данных. Если используется кэширование временных объектов, то SQL Server вместо удаления таблицы (DROP) опустошает ее (TRUNCATE). Для каждого индекса остаются выделенными две страницы: одна карта размещения индексов (IAM) и одна страница данных. При следующем создании таблицы SQL Server повторно использует эти страницы, благодаря чему требуется меньше изменений в картах размещения и системных каталогах.

Рассмотрим листинг 9.5, где определена хранимая процедура, которая создает и удаляет временную таблицу.

Листинг 9.5. Кэширование временных объектов: хранимая процедура

```
USE tempdb
GO

CREATE PROC dbo.TempTableCaching
AS
    CREATE TABLE #T(C INT NOT NULL PRIMARY KEY);
    DROP TABLE #T;
```

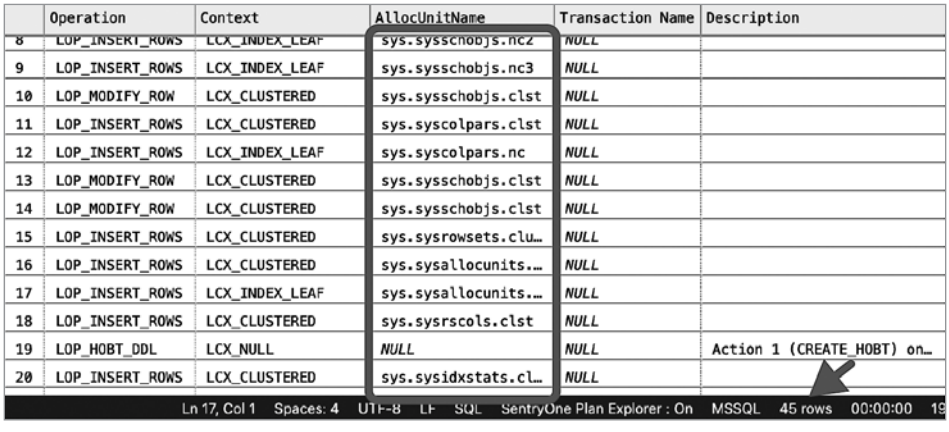
Запустите эту хранимую процедуру и изучите созданные ею записи журнала транзакций. Код приведен в листинге 9.6.

Листинг 9.6. Кэширование временных объектов: запуск хранимой процедуры

```
CHECKPOINT;
GO

EXEC dbo.TempTableCaching;

SELECT
    Operation, Context, AllocUnitName
    ,[Transaction Name], [Description]
FROM
    sys.fn_dblog(null, null);
```

Вывод кода показан на рис. 9.4. Первый вызов хранимой процедуры создал 45 записей журнала, большинство из которых относится к обновлению страниц карты распределения и системных таблиц при создании временной таблицы.

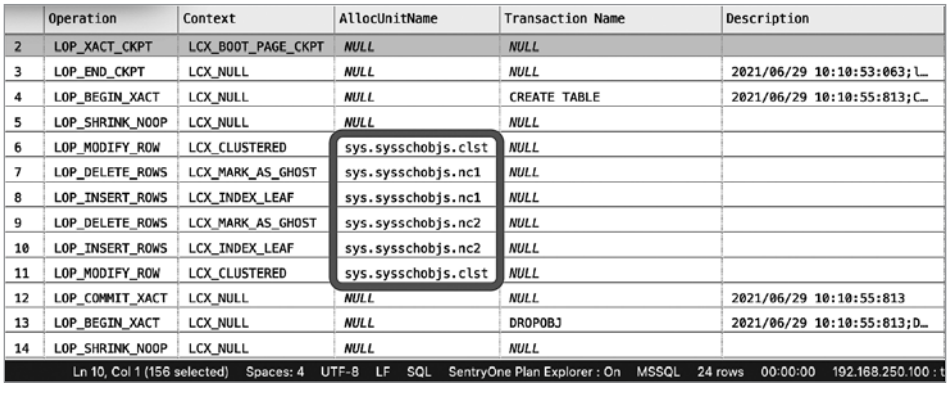
Ситуация меняется, если запустить код из листинга 9.6 второй раз. Теперь, когда временная таблица кэширована, создание таблицы добавляет в журнал всего несколько записей: все они относятся к системной таблице и не задействуют страницы карты распределения. Это видно на рис. 9.5.



	Operation	Context	AllocUnitName	Transaction Name	Description
8	LOP_INSERT_ROWS	LCX_INDEX_LEAF	sys.sysschobjs.nc2	NULL	
9	LOP_INSERT_ROWS	LCX_INDEX_LEAF	sys.sysschobjs.nc3	NULL	
10	LOP_MODIFY_ROW	LCX_CLUSTERED	sys.sysschobjs.clst	NULL	
11	LOP_INSERT_ROWS	LCX_CLUSTERED	sys.syscolpars.clst	NULL	
12	LOP_INSERT_ROWS	LCX_INDEX_LEAF	sys.syscolpars.nc	NULL	
13	LOP_MODIFY_ROW	LCX_CLUSTERED	sys.sysschobjs.clst	NULL	
14	LOP_MODIFY_ROW	LCX_CLUSTERED	sys.sysschobjs.clst	NULL	
15	LOP_INSERT_ROWS	LCX_CLUSTERED	sys.sysrowsets.clu…	NULL	
16	LOP_INSERT_ROWS	LCX_CLUSTERED	sys.sysallocunits.…	NULL	
17	LOP_INSERT_ROWS	LCX_INDEX_LEAF	sys.sysallocunits.…	NULL	
18	LOP_INSERT_ROWS	LCX_CLUSTERED	sys.sysrscols.clst	NULL	
19	LOP_HOBT_DDL	LCX_NULL	NULL	NULL	Action 1 (CREATE_HOBT) on…
20	LOP_INSERT_ROWS	LCX_CLUSTERED	sys.sysidxstats.cl…	NULL	

Ln 17, Col 1 Spaces: 4 UTF-8 LF SQL SentryOne Plan Explorer : On MSSQL 45 rows 00:00:00 19

Рис. 9.4. Кэширование временных объектов: записи журнала после первого вызова



	Operation	Context	AllocUnitName	Transaction Name	Description
2	LOP_XACT_CKPT	LCX_BOOT_PAGE_CKPT	NULL	NULL	
3	LOP_END_CKPT	LCX_NULL	NULL	NULL	2021/06/29 10:10:53:063;L…
4	LOP_BEGIN_XACT	LCX_NULL	NULL	CREATE TABLE	2021/06/29 10:10:55:813;C…
5	LOP_SHRINK_NOOP	LCX_NULL	NULL	NULL	
6	LOP_MODIFY_ROW	LCX_CLUSTERED	sys.sysschobjs.clst	NULL	
7	LOP_DELETE_ROWS	LCX_MARK_AS_GHOST	sys.sysschobjs.nc1	NULL	
8	LOP_INSERT_ROWS	LCX_INDEX_LEAF	sys.sysschobjs.nc1	NULL	
9	LOP_DELETE_ROWS	LCX_MARK_AS_GHOST	sys.sysschobjs.nc2	NULL	
10	LOP_INSERT_ROWS	LCX_INDEX_LEAF	sys.sysschobjs.nc2	NULL	
11	LOP_MODIFY_ROW	LCX_CLUSTERED	sys.sysschobjs.clst	NULL	
12	LOP_COMMIT_XACT	LCX_NULL	NULL	NULL	2021/06/29 10:10:55:813
13	LOP_BEGIN_XACT	LCX_NULL	NULL	DROPOBJ	2021/06/29 10:10:55:813;D…
14	LOP_SHRINK_NOOP	LCX_NULL	NULL	NULL	

Ln 10, Col 1 (156 selected) Spaces: 4 UTF-8 LF SQL SentryOne Plan Explorer : On MSSQL 24 rows 00:00:00 192.168.250.100 : 6

Рис. 9.5. Кэширование временных объектов: записи журнала после второго вызова

Кэширование временных объектов включено по умолчанию для всех временных объектов, созданных в хранимых процедурах и триггерах (объекты уровня сеанса не кэшируются). Но при этом есть несколько ограничений.

- Таблица должна быть меньше 8 Мбайт. Большие таблицы не кэшируются.
- Не должно быть инструкций DDL, которые изменяют структуру таблицы. Любая модификация схемы, за исключением `DROP TABLE`, предотвращает кэширование временных объектов. Однако если создавать встроенные индексы и ограничения в операторе `CREATE TABLE`, то SQL Server будет их кэшировать.

- В таблице не должно быть именованных ограничений. Безымянные ограничения не мешают кэшированию.

Учитывайте эти факторы при написании кода. Кэширование временных объектов помогает повысить производительность.

Параметры с табличным значением

SQL Server позволяет определять табличные типы в базе данных. Когда вы объявляете переменную табличного типа в коде, она работает так же, как табличная переменная. Переменные табличных типов также можно передавать в качестве параметров модулям T-SQL. Эти параметры называются *параметрами с табличным значением*.

На внутреннем уровне параметры с табличным значением реализуются как табличные переменные и наследуют все их преимущества и ограничения — в первую очередь отсутствие статистики и потенциально ошибочную оценку количества элементов. Кроме того, параметры доступны только для чтения. Нельзя модифицировать (вставлять, обновлять или удалять) данные из параметров с табличным значением в модулях, которым они передаются.

В листинге 9.7 показано, как использовать параметры с табличным значением. (Это просто пример возможного использования, а не эталонная реализация на все случаи жизни!) Код также демонстрирует, что табличные переменные не учитывают транзакции. С помощью этих переменных можно передавать информацию за пределы транзакции, которую вы откатываете.

Еще одно замечание: код показывает, как параметры с табличным значением используются в динамическом SQL. Очевидно, что передавать данные между модулями T-SQL можно и без динамического SQL.

Листинг 9.7. Использование параметров с табличным значением

```
CREATE TYPE dbo.tvpTransfers AS TABLE
(
    FromAccount BIGINT NOT NULL,
    ToAccount BIGINT NOT NULL,
    ADate DATETIME2(0) NOT NULL,
    Amount MONEY NOT NULL,
    PRIMARY KEY(FromAccount,ToAccount)
);
GO

CREATE PROC dbo.ProcessRejectedTransfers
(
    @RejectedTransfers dbo.tvpTransfers READONLY
)
AS
```

```
    SELECT FromAccount, ToAccount, ADate, Amount
    FROM @RejectedTransfers;
GO

CREATE PROC dbo.DoTransfers
(
    @Transfers dbo.tvpTransfers READONLY
)
AS
    DECLARE
        @RejectedTransfers dbo.tvpTransfers

    BEGIN TRAN
        INSERT INTO @RejectedTransfers
        (FromAccount, ToAccount, ADate, Amount)
            SELECT FromAccount, ToAccount, ADate, Amount
            FROM @Transfers
            WHERE Amount > 10000;
        /* ... */
    ROLLBACK -- Табличные переменные не учитывают транзакции
    EXEC sp_executesql
        N'EXEC dbo.ProcessRejectedTransfers @Transfers;'
        ,N'@Transfers dbo.tvpTransfers READONLY'
        ,@Transfers = @RejectedTransfers;
GO

DECLARE
    @Transfers dbo.tvpTransfers

INSERT INTO @Transfers
    (FromAccount, ToAccount, ADate, Amount)
VALUES
    (1,2,'2021-08-01',100)
    ,(3,4,'2021-08-02',15000)
    ,(5,6,'2021-08-03',20000);

EXEC dbo.DoTransfers @Transfers;
```

Параметры с табличным значением — один из самых быстрых способов передать пакет строк из клиентского приложения в подпрограмму T-SQL. Они на порядок быстрее, чем отдельные инструкии DML, а в некоторых случаях даже быстрее массовых операций.

По возможности используйте параметры с табличным значением, потому что они могут существенно улучшить производительность.

Обычные таблицы в базе данных tempdb и протоколирование транзакций

В `tempdb` можно создавать и использовать обычные таблицы. Они будут видны всем сеансам и будут вести себя так же, как в других базах данных. Правда,

при перезапуске SQL Server эти таблицы исчезнут, потому что при этом tempdb создается заново. Чтобы сохранить таблицы, есть два варианта: можно создать их заново, определив их в базе данных model, а можно использовать хранимые процедуры запуска.

Базу tempdb удобно использовать в качестве промежуточной области для процессов ETL (extract, transform, load — извлечение, преобразование, загрузка) в случаях, когда нужно загрузить и обработать большой объем данных *и* у процесса нет требований высокой доступности (HA, high availability). В tempdb используются модель восстановления SIMPLE и более эффективное протоколирование транзакций. Ей не требуется поддерживать восстановление после сбоя и, следовательно, хранить часть REDO записи журнала транзакций. (Эти данные позволяют SQL Server повторно применять изменения из зафиксированных транзакций, если происходит сбой до контрольной точки.)

Протоколирование транзакций в локальных временных таблицах работает еще эффективнее. Эти таблицы видны в рамках одного-единственного сеанса, что позволяет SQL Server чаще использовать операции с минимальным протоколированием (я расскажу об этом подробнее в главе 11). Такие операции, в свою очередь, повышают производительность ETL. Однако помните, что временные таблицы исчезнут, если клиент потеряет соединение с базой данных.

В табл. 9.1 показаны примеры операций с минимальным протоколированием для обычных и временных таблиц в tempdb. На эти примеры можно ориентироваться, чтобы ускорить процессы ETL и начальную загрузку данных в таблицы. Как я уже говорил, табличным переменным нужно еще меньше протоколирования, чем временным таблицам; однако их использование чревато ошибочной оценкой количества элементов во время оптимизации запроса.

Таблица 9.1. Операции с минимальным протоколированием в tempdb

Операция	С минимальным протоколированием?
`SELECT INTO dbo.RegularTable SELECT INTO #tempTable`	Да, но я не рекомендую использовать такую конструкцию, потому что при этом в таблице не создаются кластеризованные индексы
`INSERT INTO dbo.RegularTable WITH (TAB LOCK) SELECT`	Да
`INSERT INTO dbo.RegularTable SELECT`	Нет
`INSERT INTO #tempTable SELECT`	Да

Использование tempdb для процессов ETL может повысить их производительность и снизить нагрузку на сервер. Но часто удается добиться лучших резуль-

татов, если использовать устойчивые или неустойчивые таблицы, оптимизированные для памяти, а также In-Memory OLTP. Рассматривая эти варианты, тщательно тестируйте их реализацию: In-Memory OLTP устроена совсем не так, как классическая подсистема хранилища.

Внутренние компоненты, использующие tempdb

Помимо объектов, созданных пользователями, сам SQL Server тоже использует базу данных `tempdb` для хранения внутренних объектов. Две наиболее распространенные категории таких объектов — это хранилище версий и внутренние наборы строк, которые генерируются операциями сортировки (Sort), хеширования (Hash) и обмена (Exchange) при переносе в `tempdb`.

Рассмотрим оба случая подробно.

Хранилище версий

Некоторые функции SQL Server опираются на управление версиями строк. Помимо оптимистичных уровней изоляции, таких как `RCSI` и `SNAPSHOT`, управление версиями строк используется при оперативном перестроении индекса, а также триггерами и множественными активными наборами результатов (MARS, multiple active result sets). Как вы узнали в главе 8, старые версии строк содержатся в хранилище версий `tempdb`.

На рис. 9.6 показано, как ведет себя хранилище версий, когда в системе выполняется большая транзакция с управлением версиями строк. Можно видеть, как растет размер хранилища версий (счетчик производительности `Version Store Size (KB)`) и из-за этого запускаются события автоувеличения `tempdb`.

В данном случае SQL Server очистил хранилище версий после завершения транзакции, однако для очистки потребовалось некоторое время. Процесс очистки работает асинхронно и выполняется по расписанию.

Рост хранилища версий — одна из распространенных причин чрезмерного разбухания `tempdb`. SQL Server не удаляет из хранилища версии, возникшие после начальной точки самой старой из активных транзакций с управлением версиями строк. Из-за незафиксированных неуправляемых транзакций может увеличиться размер хранилища версий и `tempdb`, даже если `tempdb` не создает версии строк сама по себе.

В листинге 9.8 показано, как определить пять самых старых транзакций с управлением версиями строк, используя представление `sys.dm_tran_active_snapshot_database_transactions`. В экстренной ситуации можно принудительно завершить сеанс, который не дает очистить хранилище версий.

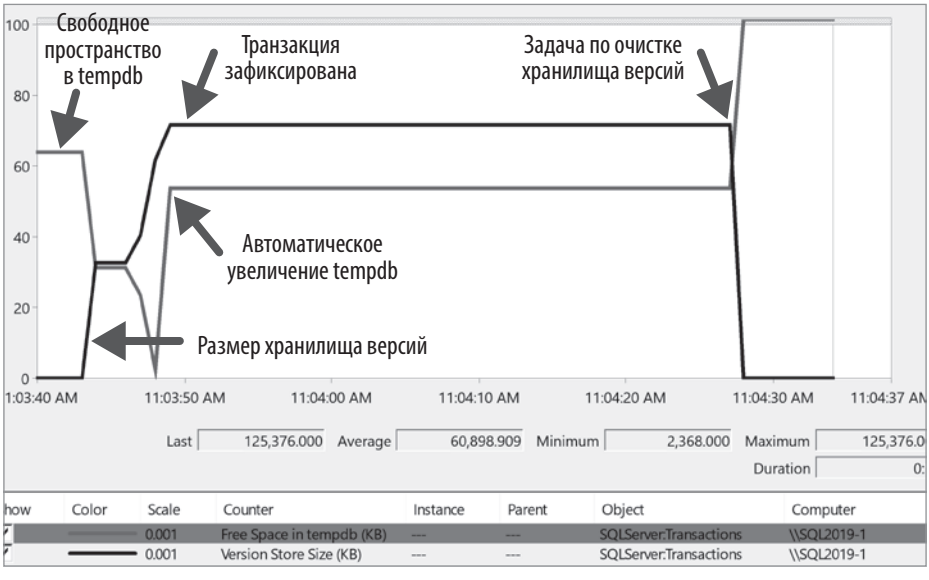


Рис. 9.6. Рост и очистка хранилища версий

Листинг 9.8. Получение пяти самых старых транзакций с управлением версиями строк

```
SELECT TOP 5
    at.transaction_id
    ,at.elapsed_time_seconds
    ,at.session_id
    ,s.login_time
    ,s.login_name
    ,s.host_name
    ,s.program_name
    ,s.last_request_start_time
    ,s.last_request_end_time
    ,er.status
    ,er.wait_type
    ,er.wait_time
    ,er.blocking_session_id
    ,er.last_wait_type
    ,st.text AS [SQL]
FROM
    sys.dm_tran_active_snapshot_database_transactions at WITH (NOLOCK)
        JOIN sys.dm_exec_sessions s WITH (NOLOCK) on
            at.session_id = s.session_id
        LEFT JOIN sys.dm_exec_requests er WITH (NOLOCK) on
            at.session_id = er.session_id
        OUTER APPLY
            sys.dm_exec_sql_text(er.sql_handle) st
ORDER BY
    at.elapsed_time_seconds DESC;
```

Длительные транзакции на доступных для чтения вторичных репликах в группах доступности могут отложить очистку хранилища версий. Я расскажу об этом в главе 12.

В объекте производительности `Transactions` есть несколько счетчиков, с помощью которых можно отслеживать поведение хранилища версий:

`Version Store Size (KB )`

Текущий размер хранилища версий (Кбайт).

`Version Generation rate (KB/s)`

Скорость роста хранилища версий (Кбайт/с).

`Version Cleanup rate (KB/s)`

Скорость очистки хранилища версий (Кбайт/с).

`Longest Transaction Running Time`

Продолжительность (в секундах) самой старой активной транзакции с управлением версиями строк.

`Free Space in tempdb (KB)`

Объем свободного места в `tempdb`. Хотя этот счетчик не связан с хранилищем версий, его можно использовать для общего мониторинга `tempdb`.

В листинге 9.9 показаны запросы, с помощью которых можно анализировать, как каждая база данных использует хранилище версий. Эти запросы могут пригодиться при устранении неполадок, связанных с чрезмерным ростом хранилища версий на серверах с несколькими базами данных.

Первый запрос будет работать в SQL Server 2016 и более поздних версиях. Второй запрос можно использовать в старых версиях, однако он более ресурсоемкий и дает чуть менее точные результаты.

Листинг 9.9. Использование хранилища версий отдельными базами данных

```
-- SQL Server 2016 SP2 и более поздние версии
SELECT
    DB_NAME(database_id) AS [DB]
    ,database_id
    ,reserved_page_count
    ,CONVERT(DECIMAL(12,3),reserved_space_kb / 1024.)
        AS [Reserved Space (MB)]
FROM
    sys.dm_tran_version_store_space_usage WITH (NOLOCK)
OPTION (RECOMPILE);
```

```
-- SQL Server 2014 и более ранние версии. Менее точные результаты
SELECT
    DB_NAME(database_id) AS [DB]
    ,database_id
    ,CONVERT(DECIMAL(12,3),
        SUM(record_length_first_part_in_bytes +
            record_length_second_part_in_bytes) / 1024. / 1024.
    ) AS [Version Store (MB)]
FROM
    sys.dm_tran_version_store WITH (NOLOCK)
GROUP BY
    database_id
OPTION (RECOMPILE, MAXDOP 1);
```

Хранилище версий — это ключевой компонент многих функций SQL Server. Следите за его размером и поведением, обращая особое внимание на длительные и неуправляемые транзакции с управлением версиями строк.

Переносы (spills)

Как вы помните из главы 7, операторам `Sort`, `Hash` и `Exchange` нужна память для хранения данных. Когда предоставленной для запроса памяти недостаточно, данные *переносятся* в `tempdb` и операции выполняются там. Переносы[1] влияют на производительность запросов, потому что доступ к данным в базе значительно медленнее, чем выполнение операции в памяти. При этом также увеличивается нагрузка на `tempdb` и может вырасти ее размер.

В последних версиях SQL Server обратная связь по предоставлению памяти может снизить количество переносов. Но проблема решена не полностью, да и то только в SQL Server Enterprise. Возможно, вам понадобится искать и оптимизировать запросы с неправильным или избыточным предоставлением памяти, используя методы из главы 7.

Чтобы выявить запросы, которые переносят данные в `tempdb`, можно использовать расширенные события `sort_warning`, `hash_warning` и `exchange_spill`. В листинге 9.10 показаны код, который создает сеанс расширенного события, и запрос, с помощью которого можно анализировать результаты.

Листинг 9.10. Использование расширенных событий для обнаружения запросов с переносом данных в `tempdb`

```
CREATE EVENT SESSION [Spills]
ON SERVER
```

[1] Перенос (spill) в документации Microsoft называют и *переносом*, и *утечкой*, и *временной записью на диск*, и *сбросом*. В официальной русской локализации Excel большими буквами `#ПЕРЕНОС!` там, где в оригинале `#SPILL!`. Думаю, этот перевод стоит считать официальной позицией Microsoft. — *Примеч. ред.*

```
ADD EVENT sqlserver.hash_warning
(
    ACTION
    (
        sqlserver.database_id
        ,sqlserver.plan_handle
        ,sqlserver.session_id
        ,sqlserver.sql_text
        ,sqlserver.query_hash
        ,sqlserver.query_plan_hash
    )
    WHERE ([sqlserver].[is_system]=0)
),
ADD EVENT sqlserver.sort_warning
(
    ACTION
    (
        sqlserver.database_id
        ,sqlserver.plan_handle
        ,sqlserver.session_id
        ,sqlserver.sql_text
        ,sqlserver.query_hash
        ,sqlserver.query_plan_hash
    )
    WHERE ([sqlserver].[is_system]=0)
),
ADD EVENT sqlserver.exchange_spill
(
    ACTION
    (
        sqlserver.database_id
        ,sqlserver.plan_handle
        ,sqlserver.session_id
        ,sqlserver.sql_text
        ,sqlserver.query_hash
        ,sqlserver.query_plan_hash
    )
    WHERE ([sqlserver].[is_system]=0)
)
ADD TARGET package0.ring_buffer;
GO

-- Начинаем сеанс расширенного события
-- Позволим ему выполняться некоторое время и собирать информацию
ALTER EVENT SESSION [Spills]
ON SERVER
STATE = START;
GO

-- Анализируем результаты
DROP TABLE IF EXISTS #tmpXML;
CREATE TABLE #tmpXML
(
```

```
    EventTime DATETIME2(7) NOT NULL,
    [Event] XML
);

DECLARE
    @TargetData XML;

SELECT
    @TargetData = CONVERT(XML,st.target_data)
FROM
    sys.dm_xe_sessions s WITH (NOLOCK)
        JOIN sys.dm_xe_session_targets st WITH(NOLOCK) ON
            s.address = st.event_session_address
WHERE
    s.name = 'Spills' and st.target_name = 'ring_buffer';

INSERT INTO #tmpXML(EventTime, [Event])
    SELECT
        t.e.value('@timestamp','datetime'), t.e.query('.')
    FROM
        @TargetData.nodes('/RingBufferTarget/event') AS t(e);

;WITH EventInfo
AS
(
    SELECT
        t.EventTime
        ,t.[Event].value('/event[1]/@name','sysname') AS [Event]
        ,t.[Event].value('(/event[1]/action[@name="session_id"]/value
                          /text())[1]'
            ,'smallint') AS [Session]
        ,t.[Event].value('(/event[1]/action[@name="database_id"]/value
                          /text())[1]'
            ,'smallint') AS [DB]
        ,t.[Event].value('(/event[1]/action[@name="sql_text"]/value/text())[1]'
            ,'nvarchar(max)') AS [SQL]
        ,t.[Event]
            .value('(/event[1]/data[@name="granted_memory_kb"]/value/text())[1]'
                ,'bigint') AS [Granted Memory (KB)]
        ,t.[Event]
            .value('(/event[1]/data[@name="used_memory_kb"]/value/text())[1]'
                ,'bigint') AS [Used Memory (KB)]
        ,t.[Event]
            .value('xs:hexBinary((/event[1]/action[@name="plan_handle"]/value
                                  /text())[1])'
                ,'varbinary(64)') AS [PlanHandle]
        ,t.[Event].value('(/event[1]/action[@name="query_hash"]/value
                          /text())[1]'
            ,'nvarchar(64)') AS [QueryHash]
        ,t.[Event]
            .value('(/event[1]/action[@name="query_plan_hash"]/value/text())[1]'
                ,'nvarchar(64)') AS [QueryPlanHash]
FROM
    #tmpXML t
```

```
)
SELECT
    ei.*, qp.query_plan
FROM
    EventInfo ei
        OUTER APPLY sys.dm_exec_query_plan(ei.PlanHandle) qp
OPTION (RECOMPILE, MAXDOP 1);
```

Результаты можно сгруппировать по хешу запроса или хешу плана, чтобы найти запросы, которые переносятся чаще всего.

SSMS и другие инструменты отображают предупреждения о переносах в планах выполнения. На рис. 9.7 показан пример в SSMS. Не игнорируйте эти предупреждения, когда настраиваете запросы.

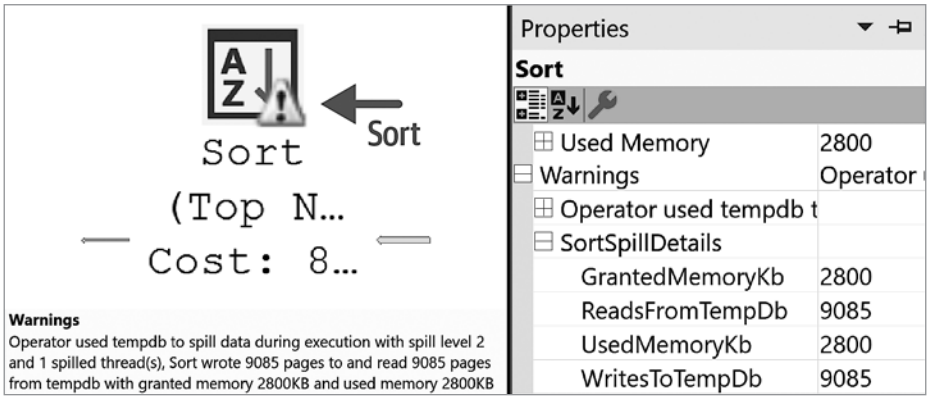


Рис. 9.7. Предупреждение сортировки в SSMS

К сожалению, полностью устранить переносы невозможно. Однако убедитесь, что критичные запросы не переносятся, и оцените, как переносы влияют на производительность `tempdb`.

Распространенные проблемы с tempdb

Проблемы с производительностью и пропускной способностью `tempdb` влияют на всю систему. Они ухудшают производительность запросов, замедляют код T-SQL и вызывают другие неполадки и побочные эффекты. К сожалению, оценить это влияние нелегко.

Обычно я начинаю с того, что смотрю на общий трафик (количество операций чтения и записи) `tempdb`, а также на метрики задержек и простоев ее файлов с помощью представления `sys.dm_io_virtual_file_stats` и кода из листинга 3.1.

Высокая скорость чтения и записи указывает на интенсивное использование tempdb. Существенная задержка может говорить как о проблемах с конфигурацией tempdb (подробнее об этом позже), так и о том, что дисковая подсистема перегружена — возможно, из-за активного трафика.

Если я вижу высокий трафик в tempdb, то пытаюсь определить его основную причину. С помощью запроса из листинга 7.1 я выясняю, чем именно занято место в tempdb. То, как объекты разных типов используют пространство, не всегда коррелирует с трафиком, который они создают. Однако, анализируя тенденции использования пространства с течением времени, часто удается получить ценную информацию.

Чтобы оценить трафик хранилища версий, можно взглянуть на счетчики производительности Version Generation rate и Version Cleanup rate. Обычно с нагрузкой хранилища ничего поделать не удается, а отключать оптимистичные уровни изоляции — не лучшее решение. Однако помните, что другие функции SQL Server — такие, как триггеры, оперативное перестроение индекса и MARS, — используют управление версиями строк. Попробуйте придумать, как уменьшить нагрузку, которую они создают.

В объектах производительности General Statistics есть два счетчика, которые помогают анализировать использование временных объектов, созданных пользователями:

Temp Tables Creation Rate

: Количество временных таблиц и табличных переменных, создаваемых в секунду.

Active Temp Tables

: Количество временных таблиц и табличных переменных, используемых в настоящее время.

Высокие значения этих счетчиков указывают на чрезмерное использование временных объектов, особенно когда при этом выходные данные листинга 9.1 показывают, что пользовательские объекты активно потребляют пространство. Стоит попытаться сократить использование временных объектов: например, для этого можно оптимизировать несколько часто выполняемых хранимых процедур.

Следующие счетчики производительности показывают, как часто в tempdb создаются некоторые типы внутренних объектов:

SQLServer:Access Methods\Worktables Created/sec

: Количество внутренних рабочих таблиц в секунду, которые SQL Server создает для поддержки буферных объектов, курсоров, больших объектов (LOB) и переменных XML.

`SQLServer:Cursor Manager By Type\Cursor worktable usage`

Количество рабочих таблиц, используемых курсорами.

К сожалению, в SQL Server нет счетчиков производительности для отслеживания переносов. Но можно использовать сеанс расширенного события, чтобы перехватывать события `sort_warning`, `hash_warning` и `exchange_spill` с целевым объектом `event_counter`, как показано в листинге 9.11.

Листинг 9.11. Подсчет количества переносов

```
CREATE EVENT SESSION [Spill_Count]
ON SERVER
ADD EVENT sqlserver.exchange_spill,
ADD EVENT sqlserver.hash_warning,
ADD EVENT sqlserver.sort_warning
ADD TARGET package0.event_counter;

-- Начинаем сеанс и позволяем ему собирать данные
ALTER EVENT SESSION [Spill_Count]
ON SERVER
STATE = START;
GO

-- Анализируем данные
DECLARE
    @TargetData XML

SELECT
    @TargetData = CONVERT(XML,st.target_data)
FROM
    sys.dm_xe_sessions s WITH (NOLOCK)
        JOIN sys.dm_xe_session_targets st WITH(NOLOCK) ON
            s.address = st.event_session_address

WHERE
    s.name = 'Spill_Count' and st.target_name = 'event_counter';

;WITH EventInfo
AS
(
    SELECT
        t.e.value('@name','sysname') AS [Event]
        ,t.e.value('@count','bigint') AS [Count]
    FROM
        @TargetData.nodes
            ('/CounterTarget/Packages/Package[@name="sqlserver"]/Event')
                AS t(e)
)
SELECT [Event], [Count]
FROM EventInfo
OPTION (RECOMPILE, MAXDOP 1);
```

Также можно добиться улучшений, если оптимизировать запросы, которые переносятся в tempdb, и сократить использование курсоров. Сосредоточьтесь на наиболее важных и часто выполняемых запросах и хранимых процедурах.

В любом случае часто можно повысить эффективность, если нарастить аппаратные мощности. В наше время это стоит дешево. Но существуют и другие распространенные проблемы с tempdb, которые стоит иметь в виду.

Состязания за доступ к системным страницам

Tempdb — это высоконагруженная база данных, в которой многочисленные сеансы постоянно создают и удаляют объекты. Во время этих операций сеансы вносят изменения в системные страницы, которые отслеживают размещение объектов и метаданные.

Чтобы защитить целостность системных страниц, SQL Server сериализует доступ к ним: в любой момент времени только один сеанс может вносить изменения. Это может привести к состязаниям и снизить пропускную способность базы данных tempdb в высоконагруженных системах.

Для этой сериализации в SQL Server применяются внутренние объекты, которые называются *кратковременными блокировками (latches)*. Они поддерживают согласованность внутренних структур данных в памяти SQL Server, позволяя только одному потоку изменять объект за раз. (Я подробно расскажу про кратковременные блокировки в главе 10).

Состязания проявляются в ожиданиях PAGELATCH, которые очень краткосрочны: их продолжительность обычно измеряется в микросекундах. Тем не менее они могут стать заметными в системах, где одновременно работают много пользователей.

Состязание за модификацию системных страниц в tempdb часто порождает ожидания PAGELATCH. Но есть и другие случаи, когда на страницах появляются кратковременные блокировки и нужно проанализировать ресурсы ожиданий, чтобы понять, связаны ли эти ожидания с tempdb. Это можно сделать с помощью кода из листинга 9.12, который регистрирует текущие ожидания PAGELATCH с помощью представления sys.dm_os_waiting_tasks.

Листинг 9.12. Регистрация ожидающих сеансов

```
-- SQL Server 2005-2017
SELECT
    wt.session_id
    ,wt.wait_type
    ,er.wait_resource
    ,er.wait_time
```

```
FROM
    sys.dm_os_waiting_tasks wt WITH (NOLOCK)
        JOIN sys.dm_exec_requests er WITH (NOLOCK) ON
            wt.session_id = er.session_id
WHERE
    wt.wait_type LIKE 'PAGELATCH%'
OPTION (MAXDOP 1, RECOMPILE);

-- SQL Server 2019 и более поздние версии
SELECT
    wt.session_id
    ,wt.wait_type
    ,er.wait_resource
    ,er.wait_time
    ,pi.database_id
    ,pi.file_id
    ,pi.page_id
    ,pi.object_id
    ,OBJECT_NAME(pi.object_id,pi.database_id) as [object]
    ,pi.index_id
    ,pi.page_type_desc
FROM
    sys.dm_os_waiting_tasks wt WITH (NOLOCK)
        JOIN sys.dm_exec_requests er WITH (NOLOCK) ON
            wt.session_id = er.session_id
        CROSS APPLY
            sys.fn_PageResCracker(er.page_resource) pc
        CROSS APPLY
            sys.dm_db_page_info(pc.db_id,pc.file_id
                ,pc.page_id,'DETAILED') pi
WHERE
    wt.wait_type LIKE 'PAGELATCH%'
OPTION (MAXDOP 1, RECOMPILE);
```

На рис. 9.8 показан пример вывода. В SQL Server 2019 есть несколько дополнительных функций, которые позволяют получить более подробную информацию. В более старых версиях SQL Server можно просмотреть столбец `wait_resource`, где первой частью ссылки на ресурс является `database_id`. Значение 2 соответствует базе `tempdb`.

Поскольку отдельные ожидания `PAGELATCH` обычно очень короткие, их легко пропустить в представлении `sys.dm_os_waiting_tasks`. Чтобы их отслеживать, можно выполнить запрос несколько раз или использовать расширенные события.

В листинге 9.13 показан сеанс расширенных событий, который регистрирует ожидания кратковременной блокировки для каждой базы данных. Этот сеанс порождает накладные расходы, поэтому не держите его включенным, когда не занимаетесь устранением неполадок. Здесь я в целях экономии ограничиваю количество собираемых событий.

	session_id	wait_type	wait_resource	wait_time	database_id	file_id	page_id	object_id	object	index_id	page_type_desc
1	60	PAGELATCH_EX	2:1:118	3	2	1	118	34	sysschobjs	2	INDEX_PAGE
2	65	PAGELATCH_EX	2:1:307	0	2	1	307	34	sysschobjs	1	DATA_PAGE
3	69	PAGELATCH_EX	2:1:307	2	2	1	307	34	sysschobjs	1	DATA_PAGE
4	70	PAGELATCH_EX	2:1:118	4	2	1	118	34	sysschobjs	2	INDEX_PAGE
5	75	PAGELATCH_EX	2:1:118	2	2	1	118	34	sysschobjs	2	INDEX_PAGE
6	76	PAGELATCH_SH	2:1:118	3	2	1	118	34	sysschobjs	2	INDEX_PAGE
7	77	PAGELATCH_EX	2:3:830	0	2	3	830	34	sysschobjs	1	DATA_PAGE
8	78	PAGELATCH_EX	2:1:118	5	2	1	118	34	sysschobjs	2	INDEX_PAGE
9	80	PAGELATCH_EX	2:3:2052	3	2	3	2052	34	sysschobjs	1	DATA_PAGE
10	81	PAGELATCH_EX	2:3:830	6	2	3	830	34	sysschobjs	1	DATA_PAGE
11	82	PAGELATCH_EX	2:3:830	3	2	3	830	34	sysschobjs	1	DATA_PAGE
12	83	PAGELATCH_EX	2:3:2052	4	2	3	2052	34	sysschobjs	1	DATA_PAGE
13	84	PAGELATCH_EX	2:3:830	5	2	3	830	34	sysschobjs	1	DATA_PAGE

Рис. 9.8. Ожидания PAGELATCH

Листинг 9.13. Регистрация ожиданий кратковременных блокировок

```
CREATE EVENT SESSION [Latch Waits] ON SERVER
ADD EVENT sqlserver.latch_suspend_end
ADD TARGET package0.ring_buffer
(SET max_events_limit=2000);
GO

-- Дальнейший код анализирует собранные результаты
DROP TABLE IF EXISTS #tmpXML;
CREATE TABLE #tmpXML
(
    EventTime DATETIME2(7) NOT NULL,
    [Event] XML
);

DECLARE
    @TargetData XML;

SELECT
    @TargetData = CONVERT(XML,st.target_data)
FROM
    sys.dm_xe_sessions s WITH (NOLOCK)
        JOIN sys.dm_xe_session_targets st WITH(NOLOCK) ON
            s.address = st.event_session_address
WHERE
    s.name = 'Latch Waits' and st.target_name = 'ring_buffer';

INSERT INTO #tmpXML(EventTime, [Event])
    SELECT t.e.value('@timestamp','datetime'), t.e.query('.')
    FROM @TargetData.nodes('/RingBufferTarget/event') AS t(e);

;WITH EventInfo
AS
(
    SELECT
        t.[EventTime] as [Time]
```

```
        ,t.[Event].value('(/event[1]/data[@name="database_id"]/value/text())[1]'
            ,'smallint') AS [DB]
        ,t.[Event].value('(/event[1]/data[@name="duration"]/value/text())[1]'
            ,'bigint') AS [Duration]
    FROM
        #tmpXML t
)
SELECT
    MONTH([Time]) as [Month]
    ,DAY([Time]) as [Day]
    ,DATEPART(hour,[Time]) as [Hour]
    ,DATEPART(minute,[Time]) as [Minute]
    ,[DB]
    ,COUNT(*) as [Latch Count]
    ,CONVERT(DECIMAL(15,3),SUM(Duration / 1000.)) as [Duration (MS)]
FROM
    EventInfo ei
GROUP BY
    MONTH([Time]),DAY([Time]),DATEPART(hour,[Time]),DATEPART(minute,[Time]),[DB]
ORDER BY
    [Month],[Day],[Hour],[Minute],[DB]
OPTION (RECOMPILE, MAXDOP 1);
```

Какие меры можно предпринять в случае состязания за доступ к системным страницам `tempdb`? В выпуске Enterprise Edition SQL Server 2019 можно воспользоваться новой функцией — *метаданными `tempdb`, оптимизированными для памяти (memory-optimized tempdb metadata)*. Эта функция преобразует системные таблицы в `tempdb` в неустойчивые таблицы без кратковременных блокировок, оптимизированные для памяти.

Чтобы включить эту функцию, выполните команду `ALTER SERVER CONFIGURATION SET MEMORY_OPTIMIZED_TEMPDB_METADATA = ON`. Чтобы изменения вступили в силу, нужно перезапустить сервер. Проверить, включена ли эта функция, можно с помощью команды `SELECT SERVERPROPERTY('IsTempdbMetadataMemoryOptimized')`.

Когда она включена, возникает несколько ограничений: в частности, нельзя создавать индексы columnstore для таблиц `tempdb`. Но функция все равно полезна, если только на вашем сервере не установлено слишком мало памяти. (Подробнее об ограничениях метаданных `tempdb`, оптимизированных для памяти, можно прочитать в документации Microsoft[1].)

К сожалению, в предыдущих версиях SQL Server сложнее обрабатывать эти состязания. Но кое-что все равно можно сделать. Сначала проверьте конфигурацию `tempdb`. Поскольку страницы выделяются на отдельные файлы, можно создать несколько файлов данных, чтобы уменьшить состязания. Правда, полезный эффект от этого будет быстро снижаться по мере роста количества файлов.

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

В старых версиях SQL Server (до 2016) включите флаг трассировки `T1118`, который отключает выделения смешанных экстентов. Это уменьшит количество изменений системных страниц во время операций выделения и освобождения.

В конце концов, возможно, придется подумать, как сократить использование `tempdb`. Чем меньше обращений к базе данных, тем ниже вероятность состязаний.

Нехватка места

Нехватка места в `tempdb` — это всегда плохо. Обычно она приводит к промышленным сбоям, например к отказу запроса, которому необходимо записать данные в базу. Это может повлиять даже на сеансы, которые явно не используют `tempdb`. Например, если у вас включены оптимистичные уровни изоляции, то инструкции `UPDATE` и `DELETE` завершатся ошибкой при попытке записи в хранилище версий.

Я не зря продолжаю повторять, что важно обеспечить мониторинг свободного места. В зависимости от ваших методов настройки и обслуживания можно отслеживать свободное место на диске, доступное пространство в базе данных или и то и другое. Настройте уведомления о том, что в `tempdb` остается критически мало места.

Имейте в виду, что `tempdb` может заполниться довольно быстро. Например, если запросы требуют сортировать сотни гигабайт данных, они перенесут данные в `tempdb` и могут моментально занять все доступное пространство. Учтите это при планировании емкости: размещайте `tempdb` на достаточно просторных дисках и настройте подходящие пороговые значения для уведомлений о мониторинге пространства.

Если вы размещаете файл журнала `tempdb` на том же диске, что и файлы данных, стоит представлять себе его размер и доступное место. Эту информацию можно получить с помощью команды `DBCCSQLPERF(LOGSPACE)` или представления `sys.database_files`. В главе 11 я приведу дополнительные сценарии и расскажу, как устранять проблемы, связанные с ростом журнала.

Есть три динамических представления, позволяющих отслеживать дисковое пространство. Первое — `sys.dm_db_file_space_usage` — содержит информацию о занятом и доступном пространстве в файлах базы данных `tempdb`. В качестве первого шага при устранении неполадок запустите код из листинга 9.1.

Методы, рассмотренные ранее в этой главе, помогут вам анализировать проблемы, связанные с ростом хранилища версий. Однако если пространство занято пользовательскими или внутренними временными объектами, то для поиска сеансов, которые занимают больше всего места, можно использовать два других представления:

`sys.dm_db_session_space_usage`[1]

Количество страниц, выделенных и освобожденных каждым сеансом в базе данных.

`sys.dm_db_task_space_usage`[2]

Информация о выделении дискового пространства для задач, выполняющихся в данный момент. Эти сведения не отображаются в представлении `sys.dm_db_session_space_usage` до тех пор, пока задача не будет завершена.

Код в листинге 9.14 объединяет данные из обоих представлений и позволяет обнаружить сеансы, которые занимают больше всего места в `tempdb`.

Листинг 9.14. Обнаружение сеансов, занимающих больше всего места в `tempdb`

```
;WITH SpaceUsagePages
AS
(
    SELECT
        ss.session_id
        ,ss.user_objects_alloc_page_count +
            ISNULL(SUM(ts.user_objects_alloc_page_count),0)
                AS [user_alloc_page_count]
        ,ss.user_objects_dealloc_page_count +
            ISNULL(SUM(ts.user_objects_dealloc_page_count),0)
                AS [user_dealloc_page_count]
        ,ss.user_objects_deferred_dealloc_page_count
                AS [user_deferred_page_count]
        ,ss.internal_objects_alloc_page_count +
            ISNULL(SUM(ts.internal_objects_alloc_page_count),0)
                AS [internal_alloc_page_count]
        ,ss.internal_objects_dealloc_page_count +
            ISNULL(SUM(ts.internal_objects_dealloc_page_count),0)
                AS [internal_dealloc_page_count]
    FROM
        sys.dm_db_session_space_usage ss WITH (NOLOCK) LEFT JOIN
            sys.dm_db_task_space_usage ts WITH (NOLOCK) ON
                ss.session_id = ts.session_id
    GROUP BY
        ss.session_id
        ,ss.user_objects_alloc_page_count
        ,ss.user_objects_dealloc_page_count
        ,ss.internal_objects_alloc_page_count
        ,ss.internal_objects_dealloc_page_count
        ,ss.user_objects_deferred_dealloc_page_count
)
```

[1] https://oreil.ly/ojd7H

[2] https://oreil.ly/mqjK8

```
,SpaceUsage
AS
(
    SELECT
        session_id
        ,CONVERT(DECIMAL(12,3),
            ([user_alloc_page_count] - [user_dealloc_page_count]) / 128.
        ) AS [user_used_mb]
        ,CONVERT(DECIMAL(12,3),
            ([internal_alloc_page_count] - [internal_dealloc_page_count]) / 128.
        ) AS [internal_used_mb]
        ,CONVERT(DECIMAL(12,3),user_deferred_page_count / 128.)
            AS [user_deferred_used_mb]
    FROM
        SpaceUsagePages
)
SELECT
    su.session_id
    ,su.user_used_mb
    ,su.internal_used_mb
    ,su.user_deferred_used_mb
    ,su.user_used_mb + su.internal_used_mb AS [space_used_mb]
    ,es.open_transaction_count
    ,es.login_time
    ,es.original_login_name
    ,es.host_name
    ,es.program_name
    ,er.status as [request_status]
    ,er.start_time
    ,CONVERT(DECIMAL(21,3),er.total_elapsed_time / 1000.) AS [duration]
    ,er.cpu_time
    ,ib.event_info as [buffer]
    ,er.wait_type
    ,er.wait_time
    ,er.wait_resource
    ,er.blocking_session_id
FROM
    SpaceUsage su
        LEFT JOIN sys.dm_exec_requests er WITH (NOLOCK) ON
            su.session_id = er.session_id
        LEFT JOIN sys.dm_exec_sessions es WITH (NOLOCK) ON
            su.session_id = es.session_id
        OUTER APPLY
            sys.dm_exec_input_buffer(es.session_id, er.request_id) ib
WHERE
    su.user_used_mb + su.internal_used_mb >= 50
ORDER BY
    [space_used_mb] DESC
OPTION (RECOMPILE);
```

Чтобы облегчить проблемы с дисковым пространством, можно принудительно завершить сеансы, которые занимают больше всего места в `tempdb`. Но, очевидно,

будет лучше выяснить, почему дисковое пространство используется так интенсивно, а затем устранить основную причину проблемы.

Конфигурация базы данных tempdb

Завершим эту главу несколькими советами по конфигурации `tempdb`. Многие вопросы конфигурации мы уже рассмотрели раньше, но повторить не помешает.

Прежде всего поместите `tempdb` на самый быстрый диск. Используйте быстрое локальное хранилище, желательно накопитель NVMe большой емкости, если есть возможность. Это позволит уменьшить задержки и увеличить пропускную способность по сравнению с сетевым хранилищем (NAS, см. главу 3).

В выпусках, отличных от Enterprise, можно даже попробовать разместить `tempdb` на RAM-диске, если на сервере достаточно памяти для этого. В Enterprise Edition обычно лучше оставить память для SQL Server.

Очевидно, следует распланировать ресурсы и выделить достаточно дискового пространства для рабочей нагрузки `tempdb`. Наладьте мониторинг свободного места и настройте уведомления в случаях, когда его остается мало. Нехватка места в `tempdb` приведет к промышленным сбоям и простоям.

Есть хитрость, которая помогает, если `tempdb` расположена на быстром накопителе с ограниченной емкостью: можно распределить файлы `tempdb` между быстрым и медленными дисками. На быстром диске заранее выделите для файлов максимально возможное пространство и отключите для них автоматическое увеличение. А файлы на медленных дисках изначально сделайте очень маленькими, но с автоувеличением.

В такой конфигурации SQL Server во время обычных операций будет отдавать предпочтение файлам на быстром диске. Но в экстремальных условиях файлы на медленных дисках начнут увеличиваться, что поможет избежать инцидентов, связанных с нехваткой места. Я пользуюсь этой конфигурацией редко, но она может быть полезна в отдельных случаях, когда быстрое хранилище достаточно велико для обычных рабочих нагрузок `tempdb`, но его не хватает для пиковых нагрузок.

Создайте несколько файлов данных. Старый совет о том, что количество файлов должно совпадать с количеством процессоров, может быть уже неактуален. Мое эмпирическое правило таково:

Если на сервере восемь или меньше ядер ЦП, создайте такое же количество файлов данных, сколько и ядер.

Если на сервере больше восьми ядер ЦП, создайте либо восемь файлов данных, либо четверть от числа ядер — в зависимости от того, что больше, — округляя

до пакетов по четыре файла. Например, на 24-ядерном сервере нужно 8 файлов данных, а на 40-ядерном — 12 файлов.

Если обнаружатся состязания или узкие места, добавляйте файлы пакетами по четыре штуки.

Задавайте для всех файлов одинаковый начальный размер и параметры автоувеличения. Если вы используете SQL Server до версии 2016, установите флаг трассировки `T1118` (чтобы отключить выделение смешанных экстентов) и, возможно, флаг трассировки `T1117` (чтобы все файлы в файловой группе увеличивались одновременно). Оба эти флага действуют на весь сервер, включая пользовательские базы. Это не вызывает проблем в случае флага `T1118`, но не включайте флаг `T1117`, если пользовательские базы данных работают с файлами неравного размера. В этом случае сначала нужно перебалансировать данные в пользовательских базах.

Наконец, если вы используете SQL Server 2019 или более поздней версии в редакции Enterprise, попробуйте включить в `tempdb` каталоги, оптимизированные для памяти, особенно если на страницах распределения `tempdb` обнаруживаются ожидания `PAGELATCH`.

Резюме

`Tempdb` — это высоконагруженная база данных, которую совместно используют все пользовательские и системные сеансы. Высокая эффективность и пропускная способность `tempdb` крайне важны для производительности сервера.

Поместите `tempdb` на самый быстрый из ваших дисков, предпочтительно на локальный накопитель NVMe большой емкости. Убедитесь, что база данных правильно сконфигурирована. Создайте несколько файлов данных, задав одни и те же параметры автоувеличения. В версиях SQL Server до 2016 включите флаги `T1118` и, возможно, `T1117`.

Если наблюдаются ожидания `PAGELATCH`, изучите, связаны ли они с `tempdb`. В SQL Server 2019 их можно смягчить, включив каталоги `tempdb`, оптимизированные для памяти. В старых версиях SQL Server проанализируйте количество файлов `tempdb`, применяйте кэширование временных объектов и сокращайте использование `tempdb`.

Устраните чрезмерную нагрузку на `tempdb`. Выполните рефакторинг кода, который злоупотребляет временными объектами, и оптимизируйте запросы, которые приводят к чрезмерным переносам.

Не забывайте, что в табличных переменных не хранится статистика, что приводит к ошибочной оценке количества элементов и неэффективным планам

запросов. Попробуйте использовать временные таблицы вместо табличных переменных.

В следующей главе я расскажу о кратковременных блокировках — объектах синхронизации, которые SQL Server использует для защиты внутренних структур в памяти.

Чек-лист устранения неполадок

- ☐ Проверить конфигурацию `tempdb` (количество файлов, параметры автоувеличения, флаг `T1118` в старых версиях SQL Server и т. д.).
- ☐ По возможности переместить базу данных `tempdb` на локальные диски NVMe большой емкости.
- ☐ Проанализировать производительность хранилища `tempdb`.
- ☐ Изучить, как `tempdb` использует дисковое пространство. Решить проблемы, если они есть.
- ☐ Выяснить, связаны ли ожидания `PAGELATCH` с `tempdb`.
- ☐ В SQL Server 2019 редакции Enterprise рассмотреть возможность включить метаданные `tempdb`, оптимизированные для памяти, особенно если наблюдаются ожидания `PAGELATCH`, связанные с `tempdb`.
- ☐ Настроить мониторинг использования места на диске, где размещена `tempdb`.

ГЛАВА 10

Кратковременные блокировки

Кратковременные блокировки (latches) — это облегченные объекты синхронизации, которые защищают согласованность внутренних структур данных SQL Server. В отличие от обычных блокировок, которые защищают согласованность данных на уровне транзакций, кратковременные блокировки предотвращают повреждение структур данных в памяти.

Эти блокировки чаще всего недолговечны и могут быть незаметны в системах с небольшими нагрузками. Однако если нагрузка растет, состязания кратковременных блокировок могут вызвать проблемы и ограничить масштабируемость и пропускную способность системы. В этой главе я расскажу, как обнаруживать и обрабатывать такие ситуации.

Начнем с обзора кратковременных блокировок, их категорий и типов. Далее поговорим о кратковременных блокировках страниц и о том, как уменьшить их состязания. В конце главы рассмотрим другие особенности работы с распространенными типами кратковременных блокировок.

Введение в кратковременные блокировки

В информатике есть концепция *взаимного исключения*, которая означает, что несколько потоков или процессов не могут выполнять критически важный код одновременно. Представьте многопоточное приложение, в котором потоки работают с общими объектами. Код, который обращается к этим объектам, часто приходится сериализовать, чтобы избежать состояния гонки, когда несколько потоков читают и обновляют объекты одновременно[1].

[1] В каждом языке разработки есть свой набор примитивов синхронизации: например, мьютексы и критические секции. В T-SQL можно использовать блокировки приложений, чтобы сериализовать доступ к коду. Здесь я их не рассматриваю, но вы можете прочитать о них в моей книге *Expert SQL Server Transctions and Locking* (Apress, 2018) или в документации Microsoft.

На внутреннем уровне SQL Server обеспечивает взаимное исключение и защищает структуры данных в памяти с помощью кратковременных блокировок. Многие путают кратковременные блокировки с обычными, ведь и те и другие влияют на конкурентный доступ и могут препятствовать одновременному доступу к одним и тем же данным. Но между ними есть тонкая разница.

Обычные блокировки обеспечивают *логическую* согласованность данных, не позволяя сеансам работать с транзакционно несогласованными данными. А кратковременные блокировки обеспечивают *физическую* согласованность структур данных в памяти: они защищают данные от повреждения, если к ним обращается несколько рабочих процессов. Кратковременные блокировки не ограничены транзакциями. Они устанавливаются, когда исполнителю нужен доступ к объекту в памяти, и снимаются после выполнения операции.

Рассмотрим ситуацию, когда нескольким сеансам нужно обновить разные строки на одной и той же странице данных. Эти сеансы не будут блокировать друг друга обычными блокировками, если только не установят несовместимые блокировки на одних и тех же строках. Зато они могут блокировать друг друга кратковременными блокировками, чтобы не позволять друг другу одновременно обновлять объект страницы данных в памяти, отчего он может повредиться.

В SQL Server существуют пять типов кратковременных блокировок. С точки зрения совместимости они похожи на обычные блокировки, но все равно не путайте их.

Кратковременная блокировка удержания (keep latch)

Кратковременная блокировка удержания (`KP`) гарантирует, что структуру, на которой удерживается блокировка, нельзя удалить. Она совместима со всеми другими типами кратковременных блокировок, кроме блокировки удаления. Этот тип похож на блокировку стабильности схемы (`Sch-S`).

Совмещаемая кратковременная блокировка (shared latch)

Совмещаемая кратковременная блокировка (`SH`) требуется, когда потоку нужно прочитать структуру данных. Такие блокировки совместимы друг с другом, а также с кратковременными блокировками удержания и обновления.

Кратковременная блокировка обновления (update latch)

Кратковременная блокировка обновления (`UP`) позволяет другим потокам читать структуру, но не обновлять ее. В некоторых ситуациях такие блокировки улучшают конкурентный доступ, подобно обычным блокировкам обновления (`U`). Кратковременные блокировки обновления совместимы с кратковременными блокировками удержания и совмеща-

емыми кратковременными блокировками, но несовместимы со всеми другими типами.

Монопольная кратковременная блокировка (exclusive latch)

Монопольная кратковременная блокировка (EX) требуется, когда поток изменяет структуру данных. Концептуально такие блокировки аналогичны обычным монопольным блокировкам (X): они несовместимы со всеми другими типами кратковременных блокировок, кроме удержания.

Кратковременная блокировка удаления (destroy latch)

Кратковременная блокировка удаления (DT) используется для удаления структуры данных. Например, она устанавливается, когда процесс отложенной записи удаляет страницу данных из буферного пула. Эти кратковременные блокировки несовместимы с другими типами.

Когда исполнитель не может установить кратковременную блокировку на структуру данных, он приостанавливает выполнение и генерирует тип ожидания, связанный с кратковременной блокировкой. Эти типы ожидания могут принадлежать к одной из трех категорий:

PAGEIOLATCH

Ожидания PAGEIOLATCH указывают на кратковременные блокировки, связанные с вводом/выводом. SQL Server использует эти блокировки и типы ожидания, когда ожидает чтения страниц данных с диска в буферный пул. Заметный процент таких ожиданий может свидетельствовать о большом количестве неоптимизированных запросов и/или неоптимальной производительности дисковой подсистемы. В главе 3 описано, как устранять неполадки в таких случаях, поэтому здесь не будем на этом останавливаться.

PAGELATCH

Ожидания PAGELATCH указывают на кратковременные блокировки, связанные с буферным пулом, которые происходят, когда потокам нужно прочитать или изменить страницы данных и карт распределения в этом пуле. Как вы знаете из главы 9, эти ожидания могут быть вызваны состязаниями в системных каталогах tempdb. Позже в этой главе я расскажу о других возможных причинах таких ожиданий.

LATCH

Ожидания LATCH — это все остальные кратковременные блокировки, не связанные с буферным пулом. Они рассматриваются позже в этой главе.

В SQL Server используются разные типы ожидания для разных типов кратковременных блокировок. Например, PAGELATCH_EX — это ожидание монопольной кратковременной блокировки (EX) на странице буферного пула. Тип ожидания

LATCH_SH означает ожидание совмещаемой кратковременной блокировки (SH) небуферного пула.

Давайте подробно рассмотрим категории кратковременных блокировок.

Кратковременные блокировки страниц

SQL Server использует кратковременные блокировки страниц (page latches), чтобы поддерживать согласованность страниц буферного пула в памяти. Когда исполнителю нужно что-то изменить на странице, он устанавливает на ней монопольную кратковременную блокировку (EX). Аналогично, когда исполнителю нужно что-то прочитать со страницы, он устанавливает на ней совмещаемую кратковременную блокировку (SH). Исполнители могут читать данные одновременно, однако в любой момент времени лишь один исполнитель может изменять страницу, причем только если к ней не обращаются другие исполнители.

Эффект от кратковременных блокировок сильно зависит от загрузки системы и характера рабочей нагрузки. SQL Server очень быстро считывает данные со страниц в памяти и записывает на них, так что в системе с небольшой нагрузкой и небольшим количеством одновременных пользователей этого можно даже не заметить. Однако состязания за кратковременные блокировки страницы усиливаются, когда становится больше активных сеансов, одновременно обращающихся к одним и тем же страницам данных.

Есть две распространенные причины состязаний за кратковременные блокировки страниц. Эти причины не исключают друг друга, и обе могут способствовать высокому проценту ожиданий PAGELATCH на сервере. При устранении неполадок надо учитывать оба фактора.

Первая причина связана с tempdb. Рабочая нагрузка на tempdb с высокой степенью конкурентности может привести к состязаниям за кратковременные блокировки на системных страницах. (В главе 9 рассказывалось, как обнаружить и смягчить этот эффект.)

Второй и наиболее распространенный случай кратковременных блокировок страниц обычно происходит в таблицах пользователей. Он называется *горячей точкой (hot spot)*. Горячие точки часто возникают в индексах с постоянно увеличивающимися (или постоянно уменьшающимися) значениями, такими как идентификаторы, последовательности или столбцы datetime, которые наполняются по мере вставки строк. Когда несколько сеансов одновременно вставляют данные в эти индексы, SQL Server размещает все строки на одной и той же странице (последней), что приводит к состязанию за кратковременные блокировки страницы, потому что исполнители начинают блокировать друг друга.

Рассмотрим гипотетический сценарий, в котором в базе данных протоколируется информация о запросах приложений. Листинг 10.1 создает несколько таблиц для хранения данных.

Не воспринимайте эту реализацию как пример из реальной жизни. На самом деле реляционные базы данных — это худшее место для хранения журналов. Здесь я использую их лишь для того, чтобы продемонстрировать состязание за кратковременные блокировки страницы.

Листинг 10.1. Создание таблиц для хранения журналов

```
CREATE TABLE dbo.WebRequests_Disk
(
    RequestId INT NOT NULL identity(1,1),
    RequestTime DATETIME2(4) NOT NULL
        CONSTRAINT DEF_WebRequests_Disk_RequestTime
        DEFAULT SYSUTCDATETIME(),
    URL VARCHAR(255) NOT NULL,
    RequestType TINYINT NOT NULL,
    ClientIP VARCHAR(15) NOT NULL,
    BytesReceived INT NOT NULL,

    CONSTRAINT PK_WebRequests_Disk
    PRIMARY KEY NONCLUSTERED(RequestId)
);

CREATE UNIQUE CLUSTERED INDEX IDX_WebRequests_Disk_RequestTime_RequestId
ON dbo.WebRequests_Disk(RequestTime,RequestId);

CREATE TABLE dbo.WebRequestHeaders_Disk
(
    RequestId INT NOT NULL,
    HeaderName VARCHAR(64) NOT NULL,
    HeaderValue VARCHAR(256) NOT NULL,
    CONSTRAINT PK_WebRequestHeaders_Disk
    PRIMARY KEY CLUSTERED(RequestId,HeaderName)
);

CREATE TABLE dbo.WebRequestParams_Disk
(
    RequestId INT NOT NULL,
    ParamName VARCHAR(64) NOT NULL,
    ParamValue NVARCHAR(256) NOT NULL,

    CONSTRAINT PK_WebRequestParams_Disk
    PRIMARY KEY CLUSTERED(RequestId,ParamName)
);
```

Теперь запустим приложение, которое вставляет данные в эти таблицы из нескольких потоков одновременно. Код приложения можно найти в сопутствующих материалах книги.

На рис. 10.1 показаны метрики с моего тестового сервера, снятые за время работы приложения. Хотя продолжительность ожиданий отдельных кратковременных блокировок невелика — всего доли миллисекунды, в совокупности они привели к очень большим ожиданиям и ограничили пропускную способность приложения. При этом загрузка процессора *не* была максимальной.

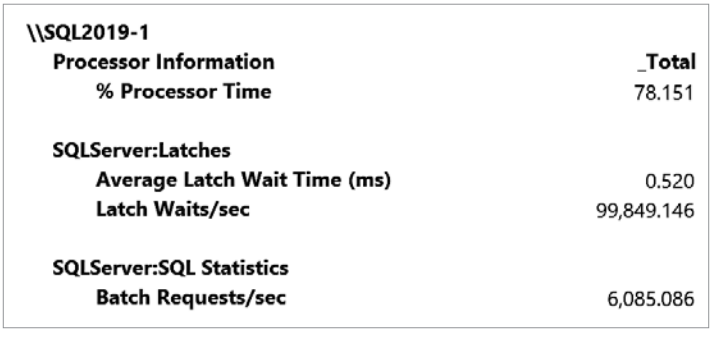


Рис. 10.1. Показатели производительности во время состязания за кратковременную блокировку страниц

На рис. 10.2 показана статистика ожидания, полученная с помощью кода из листинга 2.1. Как видите, ожидания `PAGELATCH_EX` и `PAGELATCH_SH` составляют более 60 % от общего времени ожиданий.

	Wait Type	Wait Count	Wait Time	Avg Wait Time	Avg Signal Wait Time	Avg Resource Wait Time	Percent	Running Percent
1	PAGELATCH_EX	1951297	749.168	0.0	0.0	0.0	45.332	45.332
2	WRITELOG	389893	488.770	1.0	0.0	1.0	29.575	74.908
3	PAGELATCH_SH	672816	303.111	0.0	0.0	0.0	18.341	93.249

Рис. 10.2. Статистика ожиданий, связанных с состязанием за кратковременную блокировку страницы

Чтобы обнаружить индексы, которые приводят к наибольшему количеству ожиданий кратковременной блокировки страницы, используйте функцию `sys.dm_db_index_operational_stats`[1]. Как можно догадаться по названию, она отслеживает операционные показатели индексов, в том числе количество кратковременных блокировок и продолжительность их ожиданий.

Код из листинга 10.2 обнаруживает индексы, которые способствуют возникновению горячих точек. Однако это очень упрощенная реализация; в главе 14 я покажу более серьезный код, который анализирует работоспособность индексов.

[1] https://oreil.ly/W5Sub

Листинг 10.2. Анализ статистики индексов кратковременных блокировок на странице

```
SELECT
    s.name + '.' + t.name as [table]
    ,i.index_id
    ,i.name as [index]
    ,SUM(os.page_latch_wait_count) AS [latch count]
    ,SUM(os.page_latch_wait_in_ms) as [latch wait (ms)]
FROM
    sys.indexes i WITH (NOLOCK) JOIN sys.tables t WITH (NOLOCK) on
        i.object_id = t.object_id
    JOIN sys.schemas s WITH (NOLOCK) ON
        t.schema_id = s.schema_id
    CROSS APPLY
        sys.dm_db_index_operational_stats
        (
            DB_ID()
            ,t.object_id
            ,i.index_id
            ,0
        ) os
GROUP BY
    s.name, t.name, i.name, i.index_id
ORDER BY
    SUM(os.page_latch_wait_in_ms) DESC;
```

На рис. 10.3 показан вывод кода из листинга 10.2. Как видите, эти данные позволяют быстро обнаруживать проблемные индексы в базе.

	table	index_id	index	latch count	latch wait (ms)
1	dbo.WebRequestHeaders_Disk	1	PK_WebRequestHeaders_Disk	2230279	933152
2	dbo.WebRequestParams_Disk	1	PK_WebRequestParams_Disk	356426	112319
3	dbo.WebRequests_Disk	1	IDX_WebRequests_Disk_Requ…	36621	7568
4	dbo.WebRequests_Disk	2	PK_WebRequests_Disk	984	722

Рис. 10.3. Результат функции `sys.dm_db_index_operational_stats`

Решение проблем с горячими точками: параметр индекса OPTIMIZE_FOR_SEQUENTIAL_KEY

К сожалению, устранять кратковременные блокировки страниц, возникающие из-за горячих точек, — непростая задача, особенно в старых версиях SQL Server. В SQL Server 2019 и более поздних версиях можно включить параметр индекса `OPTIMIZE_FOR_SEQUENTIAL_KEY`. Эта настройка улучшает пропускную способность в ситуации с горячими точками, но не решает проблему полностью.

Давайте включим этот параметр, как показано в листинге 10.3, и повторим тест. Я также рекомендую очистить статистику ожидания перед тем, как перезапускать приложение.

Листинг 10.3. Включение параметра `OPTIMIZE_FOR_SEQUENTIAL_KEY` (SQL Server 2019 и более поздние версии)

```
ALTER INDEX PK_WebRequestHeaders_Disk
ON dbo.WebRequestHeaders_Disk
SET (OPTIMIZE_FOR_SEQUENTIAL_KEY = ON);

ALTER INDEX PK_WebRequestParams_Disk
ON dbo.WebRequestParams_Disk
SET (OPTIMIZE_FOR_SEQUENTIAL_KEY = ON);
```

На рис. 10.4 показаны метрики производительности, когда параметр `OPTIMIZE_FOR_SEQUENTIAL_KEY` включен. Как видите, он улучшает пропускную способность, хотя ожидания кратковременных блокировок по-прежнему остаются значительными.

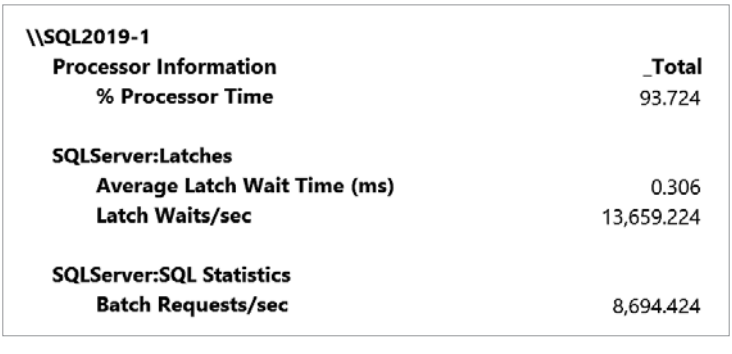


Рис. 10.4. Метрики производительности с включенным параметром `OPTIMIZE_FOR_SEQUENTIAL_KEY`

На рис. 10.5 показана статистика ожидания. В ней отображается новый тип ожидания — `BTREE_INSERT_FLOW_CONTROL`. Этот тип специфичен для реализации `OPTIMIZE_FOR_SEQUENTIAL_KEY` и на практике маскирует ожидания `PAGELATCH` и конкуренцию за кратковременные блокировки в статистике ожидания.

	Wait Type	Wait Count	Wait Time	Avg Wait Time	Avg Signal Wait Time	Avg Resource Wait Time	Percent	Running Percent
1	WRITELOG	609044	1020.615	1.0	0.0	1.0	60.430	60.430
2	BTREE_INSERT_FLOW_CONTROL	1477495	554.722	0.0	0.0	0.0	32.845	93.275
3	PAGELATCH_EX	1128697	99.052	0.0	0.0	0.0	5.865	99.140

Рис. 10.5. Статистика ожидания с включенным параметром `OPTIMIZE_FOR_SEQUENTIAL_KEY`

Хотя параметр `OPTIMIZE_FOR_SEQUENTIAL_KEY` может улучшить пропускную способность в случае горячих точек, он не избавляет полностью от кратковременных блокировок. Более того, эта настройка недоступна в старых версиях SQL Server (до 2019). В этих случаях мало что можно поделать, помимо рефакторинга схемы базы данных и приложений.

Имеет смысл обнаружить и проанализировать индексы, которые вносят наибольший вклад в кратковременные блокировки. Изучите использование индекса (я поделюсь несколькими полезными приемами в главе 14) и подумайте, нельзя ли удалить проблемные индексы или изменить их так, чтобы они перестали постоянно разрастаться, усугубляя горячие точки.

Кластеризованные индексы, определенные для столбцов идентификаторов в активных таблицах, — один из самых распространенных источников горячих точек. Чтобы уменьшить проблемы, используйте кластеризованные индексы, которые распределяют операции вставки по таблице.

Решение проблем с горячими точками: хеш-разбиение

Один из методов, позволяющих распределять вставки по таблице, называется *хеш-разбиением*. Можно разбить таблицу на разделы и использовать разные хеш-значения, чтобы распределить строки по нескольким разделам.

В листинге 10.4 показан пример кода, который переопределяет две таблицы из листинга 10.1, разбивая их на разделы с помощью нового столбца `HashVal`, вычисляемого как `RequestId % 16`. В результате данные распределяются по 16 разделам, что уменьшает частоту вставок и состязания за кратковременные блокировки в каждом отдельном разделе.

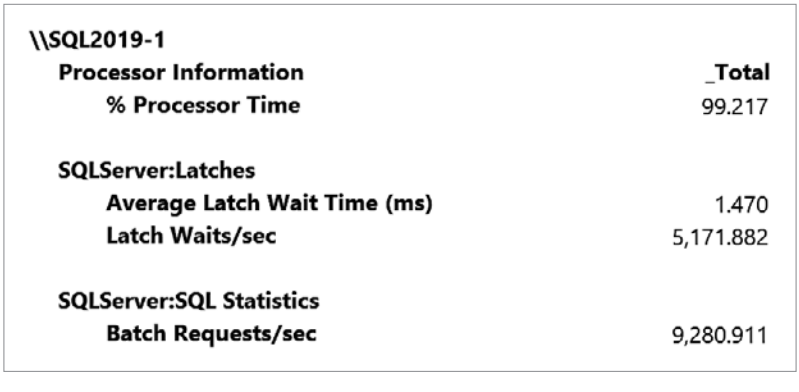
Обратите внимание, что столбец `HashVal` определяется как крайний правый столбец в индексах, чтобы сохранить порядок сортировки в каждом отдельном разделе.

Листинг 10.4. Реализация хеш-разбиения

```
-- В демонстрационных целях
TRUNCATE TABLE dbo.WebRequests_Disk;
DROP TABLE dbo.WebRequestHeaders_Disk;
DROP TABLE dbo.WebRequestParams_Disk;
GO

CREATE PARTITION FUNCTION pfHash(int)
AS RANGE LEFT FOR VALUES
(0,1,2,3,4,5,6,7,8,10,11,12,13,14,15);
CREATE PARTITION SCHEME psHash
AS PARTITION pfHash
ALL TO ([PRIMARY]);
```

```
CREATE TABLE dbo.WebRequestHeaders_Disk
(
    RequestId INT NOT NULL,
    HeaderName VARCHAR(64) NOT NULL,
    HeaderValue VARCHAR(256) NOT NULL,
    HashVal AS RequestId % 16 PERSISTED,

    CONSTRAINT PK_WebRequestHeaders_Disk
    PRIMARY KEY CLUSTERED(RequestId,HeaderName,HashVal)
    ON psHash(HashVal)
);

CREATE TABLE dbo.WebRequestParams_Disk
(
    RequestId INT NOT NULL,
    ParamName VARCHAR(64) NOT NULL,
    ParamValue nVARCHAR(256) NOT NULL,
    HashVal AS RequestId % 16 PERSISTED,

    CONSTRAINT PK_WebRequestParams_Disk
    PRIMARY KEY CLUSTERED(RequestId,ParamName,HashVal)
    ON psHash(HashVal)
);
```

На рис. 10.6 показаны метрики производительности на моем тестовом сервере, где реализовано разбиение хешей. Скорость вставки немного выше, чем при использовании `OPTIMIZE_FOR_SEQUENTIAL_KEY`, хотя разница относительно невелика. Естественно, ваши данные могут отличаться в зависимости от сценариев использования и рабочих нагрузок.



\\SQL2019-1	
Processor Information	**_Total**
% Processor Time	99.217
SQLServer:Latches	
Average Latch Wait Time (ms)	1.470
Latch Waits/sec	5,171.882
SQLServer:SQL Statistics	
Batch Requests/sec	9,280.911

Рис. 10.6. Показатели производительности с хеш-разбиением на разделы

На рис. 10.7 показана статистика ожидания. Как видите, процент ожиданий кратковременной блокировки страницы стал ниже, чем без разбиения (как было показано на рис. 10.2).

	Wait Type	Wait Count	Wait Time	Avg Wait Time	Avg Signal Wait Time	Avg Resource Wait Time	Percent	Running Percent
1	WRITELOG	331696	722.286	2.0	0.0	1.0	66.211	66.211
2	PAGELATCH_EX	1248937	291.905	0.0	0.0	0.0	26.759	92.970
3	PAGELATCH_SH	246413	68.703	0.0	0.0	0.0	6.298	99.268

Рис. 10.7. Статистика ожидания с хеш-разбиением на разделы

Хотя хеш-разбиение помогает уменьшить состязания за кратковременные блокировки, оно представляет определенную опасность. Как и при любом разбиении, данные будут распределены по нескольким внутренним таблицам (разделам). От этого изменятся планы выполнения и может ухудшиться производительность запросов.

С точки зрения уменьшения состязаний хеш-разбиение дает наилучшие результаты, когда количество разделов равно количеству логических ядер ЦП на сервере. Однако при большем количестве разделов возрастает риск снижения производительности. Помните об этом, когда разрабатываете стратегию борьбы с горячими точками, и тщательно тестируйте систему перед внедрением. Я считаю хеш-разбиение крайней мерой, если больше ничто не помогает справиться с состязаниями за кратковременные блокировки страниц.

Иногда удается сгладить побочные эффекты хеш-разбиения, используя промежуточные таблицы. В этом случае приложение может вставлять данные в промежуточные таблицы с хеш-разделами, а другой процесс будет запускаться по расписанию и копировать данные из промежуточных таблиц в основные. Хотя это решение добавляет накладные расходы на управление промежуточными таблицами, оно защищает от непредвиденного снижения производительности, которое может произойти из-за хеш-разбиения.

Решение проблем с горячими точками: In-Memory OLTP

Обсуждая проблемы производительности, связанные с кратковременными блокировками, нельзя не упомянуть In-Memory OLTP. В конце концов, одна из основных целей этой технологии — преодолеть проблемы с кратковременными и обычными блокировками, которые характерны для дисковых таблиц при больших параллельных нагрузках. Таблицы, оптимизированные для памяти, не используют блокировок и прекрасно масштабируются при параллельных рабочих нагрузках OLTP.

Но у этих преимуществ есть своя цена. In-Memory OLTP и таблицы, оптимизированные для памяти, ведут себя не так, как классические подсистемы хранения и дисковые таблицы. Важно тщательно спроектировать систему и грамотно использовать технологию, чтобы избежать побочных эффектов и проблем. Как и в случае хеш-разбиения, производительность может ухудшиться, если вы просто преобразуете дисковые таблицы в таблицы,

оптимизированные для памяти, и не будете учитывать разницу в устройстве технологий.

К счастью, таблицы, оптимизированные для памяти, идеально подходят в качестве промежуточных. При этом можно полностью избавиться от горячих точек и состязаний за кратковременные блокировки, а также уменьшить влияние побочных эффектов, связанных с In-Memory OLTP. Это эффективный вариант решения проблем с горячими точками.

Однако повторю свое предостережение: перед тем как использовать In-Memory OLTP, нужно хорошо разбираться в том, как правильно развернуть и поддерживать эту технологию. Почитайте мою книгу «Expert SQL Server In-Memory OLTP» (Apress, 2017), чтобы узнать больше.

Другие типы кратковременных блокировок

Третья категория кратковременных блокировок не связана с буферным пулом и проявляется в форме ожиданий LATCH общего вида. Подобно статистике ожидания, информацию об отдельных типах кратковременных блокировок можно получить из представления `sys.dm_os_latch_stats`[1], как показано в листинге 10.5. Можно также очистить статистику кратковременных блокировок с помощью команды `DBCC SQLPERF('sys.dm_os_latch_stats', CLEAR)`.

Листинг 10.5. Анализ статистики кратковременных блокировок

```
;WITH Latches
AS
(
    SELECT
        latch_class, wait_time_ms, waiting_requests_count
        ,100. * wait_time_ms / SUM(wait_time_ms) OVER() AS Pct
        ,100. * SUM(wait_time_ms) OVER(ORDER BY wait_time_ms DESC) /
            NULLIF(SUM(wait_time_ms) OVER(), 0) AS RunningPct
        ,ROW_NUMBER() OVER(ORDER BY wait_time_ms DESC) AS RowNum
    FROM
        sys.dm_os_latch_stats WITH (NOLOCK)
    WHERE
        wait_time_ms > 0 AND
        latch_class NOT IN (N'BUFFER',N'SLEEP_TASK')
)
SELECT
    l1.latch_class AS [Latch Type]
    ,l1.waiting_requests_count AS [Latch Count]
    ,CONVERT(DECIMAL(12,3), l1.wait_time_ms / 1000.0)
        AS [Wait Time]
```

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

```
    ,CONVERT(DECIMAL(12,1), l1.wait_time_ms / l1.waiting_requests_count)
        AS [Avg Wait Time]
    ,CONVERT(DECIMAL(6,3), l1.Pct)
        AS [Percent]
    ,CONVERT(DECIMAL(6,3), l1.RunningPct)
        AS [Running Percent]
FROM
    Latches l1
WHERE
    l1.RunningPct <= 99 OR l1.RowNum = 1
ORDER BY
    l1.RunningPct
OPTION (RECOMPILE, MAXDOP 1);
```

На рис. 10.8 показан вывод этого сценария на одном из серверов.

	Latch Type	Latch Count	Wait Time	Avg Wait Time	Percent	Running Percent
1	ACCESS_METHODS_DATASET_PARENT	3858110729	1249807.056	0.0	47.031	47.031
2	NESTING_TRANSACTION_FULL	766539824	972751.713	1.0	36.605	83.635
3	TRACE_CONTROLLER	38694619	270191.197	6.0	10.167	93.803

Рис. 10.8. Статистика кратковременных блокировок

К сожалению, типы кратковременных блокировок плохо документированы. Часто приходится копаться в нескольких источниках, чтобы понять значение каждой блокировки. Тем не менее рассмотрим несколько распространенных типов, с которыми вы можете столкнуться.

Кратковременные блокировки, связанные с параллелизмом

Существует несколько типов кратковременных блокировок, связанных с параллелизмом. Наиболее распространенные из них — `ACCESS_METHOD_DATASET_PARENT`, `ACCESS_METHODS_SCAN_RANGE_GENERATOR`, `ACCESS_METHODS_SCAN_KEY_GENERATOR` и `NESTING_TRANSATION_FULL`. Мой опыт говорит, что эти блокировки обычно появляются вверху списка в выводе представления `sys.dm_os_latch_stats`.

Обращайтесь с этими кратковременными блокировками так же, как с ожиданиями параллелизма (`CXPACKET`, `CXCONSUMER` и `EXCHANGE`), которые часто проявляются вместе с блокировками, относящимися к параллелизму. Чтобы устранить неполадки, проанализируйте и настройте использование параллелизма, как описано в главе 6.

`LOG_MANAGER`

Тип кратковременной блокировки `LOG_MANAGER` указывает на рост журнала транзакций. Такой тип появляется в ситуациях, когда что-то регулярно мешает усечению журнала. Для устранения неполадок изучите столбец

`log_reuse_wait_desc` в представлении `sys.databases` (подробнее об этом в главе 11).

Я также сталкивался с этим типом кратковременной блокировки в системах, где журнал транзакций сжимался после каждого резервного копирования журнала. Это плохая практика, потому что во время увеличения файл журнала инициализируется нулями.

`ACCESS_METHODS_HOBT_VIRTUAL_ROOT`

Кратковременная блокировка `ACCESS_METHODS_HOBT_VIRTUAL_ROOT` используется при доступе к метаданным индекса. Значительное число таких блокировок указывает на большое количество разбиений корневых страниц в индексах на основе B-деревьев. Обычно это происходит в небольших индексах с очень изменчивыми данными: например, в таблицах, которые работают как внутренние очереди в базе данных.

Индексы, которые могут привести к проблемам, можно обнаружить с помощью столбцов `tree_page_latch_wait_count` и `tree_page_latch_wait_time_ms` в представлении `sys.db_db_index_operational_stats` (подробнее в главе 14).

`ACCESS_METHODS_HOBT_COUNT`

Кратковременная блокировка `ACCESS_METHODS_HOBT_COUNT` используется, чтобы обновлять информацию о количестве страниц и строк в метаданных таблицы. Состязания за эту блокировку свидетельствуют о множестве параллельных изменений данных в некоторых таблицах.

В главе 14 мы обсудим, как отслеживать количество изменений в индексах с помощью представления `sys.db_db_index_usage_stats`. Это поможет вам обнаруживать таблицы с большим количеством модификаций. Однако может оказаться целесообразнее не использовать представления SQL Server, а связаться с разработчиками, потому что для устранения состязаний такого рода обычно требуется доработать само приложение.

`FGCB_ADD_REMOVE`

Кратковременная блокировка `FGCB_ADD_REMOVE` возникает при добавлении, удалении, увеличении и сжатии файлов в файловой группе. Проверьте, что в конфигурации базы данных включена мгновенная инициализация файлов (*Instant File Initialization*) и отключено автоматическое сжатие (*Auto Shrink*). Также убедитесь, что при автоувеличении файлы не увеличиваются слишком маленькими фрагментами.

Эта кратковременная блокировка также появляется в базах данных с очень большим количеством файлов данных, особенно в `tempdb`. В этом случае, чтобы решить проблему, пересмотрите конфигурацию базы.

`TRACE_CONTROLLER`

Как можно догадаться по названию, кратковременная блокировка `TRACE_CONTROLLER` указывает на большое количество трассировок. Обнаружив эту блокировку, пересмотрите свою стратегию мониторинга и удалите ненужный мониторинг.

От кратковременных блокировок, как и от ожиданий, нельзя избавиться полностью. Они совершенно естественны в малых дозах, но когда их становится слишком много — это признак проблем в системе. Однако не совершайте поспешных действий, а сперва выявите и устраните первопричины проблем!

Резюме

Кратковременные блокировки — это облегченные объекты синхронизации, которые защищают согласованность внутренних структур данных SQL Server. Они чаще всего недолговечны и могут быть незаметны в системах с небольшими нагрузками. Однако, если нагрузка растет, состязания кратковременных блокировок могут вызвать проблемы и ограничить масштабируемость и пропускную способность системы.

Существуют три категории кратковременных блокировок. Каждому типу блокировки (совмещаемая, монопольная и т. д.) соответствует тип ожидания в статистике ожидания.

Ожидания `PAGEIOLATCH` происходят, когда SQL Server ожидает чтения страницы данных в буферный пул. Значительный процент этих ожиданий требует устранения неполадок загрузки дисковой подсистемы (см. главу 3).

Кратковременные блокировки страниц и соответствующие ожидания `PAGELATCH` происходят, когда несколько рабочих процессов одновременно обращаются к страницам данных и обновляют их в памяти. Обычно эти ожидания инициируются состязаниями системных объектов `tempdb` или горячими точками в постоянно растущих индексах в пользовательских базах данных.

В SQL Server 2019 и более поздних версиях эффект горячих точек можно уменьшить, если включить параметр индекса `OPTIMIZE_FOR_SEQUENTIAL_KEY`. Однако во многих случаях приходится удалять или изменять индекс, а также рассматривать обходные пути с хеш-разбиением и/или промежуточными таблицами.

Кратковременные блокировки, не связанные с буферным пулом, проявляются как типы ожидания `LATCH`. Статистику этих блокировок можно получить с помощью представления `sys.dm_os_latch_stats`. Устраняя неполадки, выявите и искорените основную причину этих проблем.

В следующей главе мы рассмотрим характерные проблемы с журналом транзакций.

Чек-лист устранения неполадок

- ☐ Проанализировать влияние кратковременных блокировок страниц с помощью ожиданий `PAGELATCH`.
- ☐ Определить, происходят ожидания `PAGELATCH` от `tempdb` или от горячих точек в пользовательских базах данных.
- ☐ Устранить состязания системных объектов в `tempdb` (глава 9).
- ☐ Выявить индексы, которые способствуют появлению горячих точек, с помощью представления `sys.dm_db_index_operational_stats`. Попробовать провести рефакторинг индексов или удалить их (подробнее об этом — в главе 14).
- ☐ Включить параметр `OPTIMIZE_FOR_SEQUENTIAL_KEY` в SQL Server 2019 и более поздних версиях.
- ☐ Если прочие методы не помогают бороться с горячими точками, рассмотреть возможность рефакторинга приложения с использованием In-Memory OLTP, промежуточных таблиц и/или хеш-разбиений.
- ☐ Просмотреть статистику кратковременных блокировок в представлении `sys.dm_os_latch_stats`. Устранить неполадки и решить проблемы, если это необходимо.

ГЛАВА 11

Журнал транзакций

У каждой базы данных в SQL Server есть ровно один журнал транзакций, реализованный как один или несколько файлов журнала в дополнение к файлам данных. В журнале транзакций хранится информация об изменениях, внесенных в базу данных, поэтому в случае неожиданного завершения работы или сбоя SQL Server журнал позволяет восстанавливать базы данных до транзакционно согласованного состояния. В журнале транзакций протоколируется каждая модификация данных в базе, и низкая задержка журнала необходима для хорошей производительности системы.

В этой главе я расскажу, как SQL Server протоколирует транзакции и как журнал транзакций работает на внутреннем уровне. Затем я продемонстрирую несколько эффективных методов настройки журнала транзакций и объясню, что делать, если он достигнет предельного размера. Наконец, разберемся, как бороться с недостаточной производительностью журнала транзакций.

Внутреннее устройство журнала транзакций

В SQL Server журнал транзакций используется для того, чтобы поддерживать каждую базу данных в *транзакционно согласованном* состоянии: все модификации данных, выполненные внутри одной и той же транзакции, должны либо фиксироваться целиком, либо откатываться целиком. SQL Server не допускает, чтобы данные были транзакционно несогласованными, и не может применить к базе данных лишь какую-то часть изменений из незафиксированной транзакции.

Журнал транзакций гарантирует согласованность. В нем хранится поток *записей журнала*, порожденных модификациями данных и некоторыми внутренними операциями. Каждая запись журнала имеет уникальный автоматически увеличивающийся *порядковый номер записи* (*LSN, log sequence number*) и описывает, в чем заключалось изменение данных. В запись входит информация о затрону-

той строке, старой и новой версиях данных, транзакции, которая выполнила модификацию, и т. д.

Каждая страница данных помнит LSN последней записи журнала, относящейся к изменению этой страницы. Во время восстановления SQL Server может сравнить LSN из записей журнала и страниц данных и выяснить, были ли самые последние изменения сохранены в файлах данных. В записи журнала хранится достаточно информации, чтобы при необходимости отменить или повторить операцию.

SQL Server ведет *журнал с опережающей записью* (*WAL, write-ahead logging*) — это гарантирует, что записи журнала всегда вносятся в файл журнала *до* того, как «грязные» страницы данных будут сохранены в базе данных. Внимательные читатели заметят, что в главе 3 я упоминал, будто записи журнала сохраняются синхронно с изменениями данных, а страницы данных — асинхронно во время процесса контрольной точки. Концептуально это так и есть, но я выскажусь точнее: SQL Server кэширует записи журнала в небольших кэшах памяти, называемых *буферами журнала (log buffers)*, а затем вносит их в файл журнала пакетами, чтобы уменьшить количество операций ввода/вывода, связанных с журналом.

У каждой базы данных есть собственный буфер журнала, состоящий из структур размером по 60 Кбайт, которые называются *блоками журнала (log blocks)*. Каждый буфер журнала (и каждая база данных) поддерживает до 128 блоков журнала. SQL Server записывает блок в файл журнала за одну операцию ввода/вывода, но он не всегда ждет, пока блок заполнится. Типичный объем операции ввода/вывода записи журнала варьируется от 512 байт до 60 Кбайт.

К сожалению, документация по SQL Server непоследовательна в своей терминологии, и в ней термины «*блок журнала*» и «*буфер журнала*» часто путаются. Просто имейте в виду, что SQL Server кэширует записи журнала в памяти перед тем, как записывать их на диск.

Модификация данных и протоколирование транзакций

Давайте подробнее рассмотрим, как SQL Server модифицирует данные. На рис. 11.1 показана база данных с пустым буфером журнала и журналом транзакций. У последней транзакции в журнале LSN равен 7314.

Предположим, что выполняются две активные транзакции: `T1` и `T2`. Записи журнала `BEGIN TRAN` для обеих этих транзакций уже сохранены в журнале и не показаны на диаграмме.

Предположим, что транзакция `T1` обновляет одну из строк на странице (1:24413). Эта операция создает новую запись журнала, которая помещается в буфер журнала. Операция также обновляет страницу данных, пометив ее как «грязную» и изменив номер LSN в заголовке страницы. Это показано на рис. 11.2.

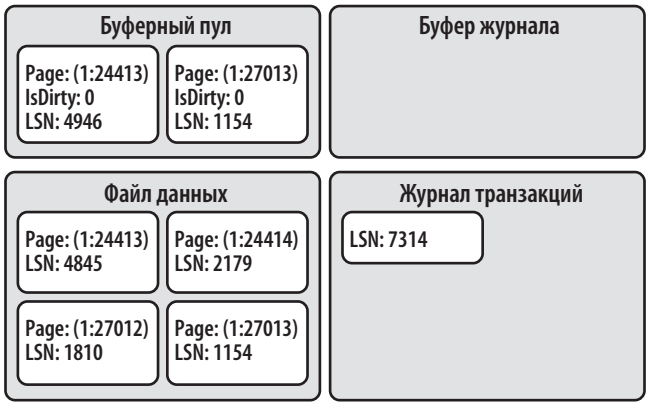


Рис. 11.1. Модификация данных и протоколирование транзакций: исходное состояние

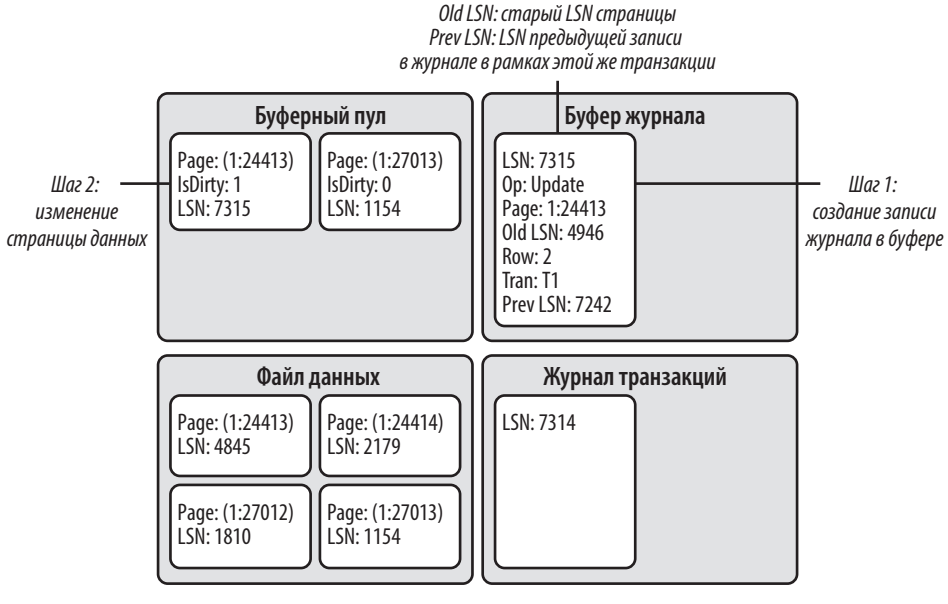


Рис. 11.2. Модификация данных и протоколирование транзакций:
состояние после первого обновления

В этот момент запись журнала еще не *закреплена (hardened)*, то есть не сохранена в файле журнала. Это не проблема, пока страница данных тоже не сохранена в файле данных. В случае сбоя SQL Server исчезнут как запись в журнале, так и изменения на странице данных. Это нормально, потому что транзакция не была зафиксирована.

Теперь предположим, что транзакция `T2` вставляет новую строку на страницу (1:27013), а транзакция `T1` удаляет другую строку на этой же странице. Эти операции создают две записи журнала, которые помещаются в буфер журнала, как показано на рис. 11.3. На этот момент все записи журнала еще находятся в буфере.

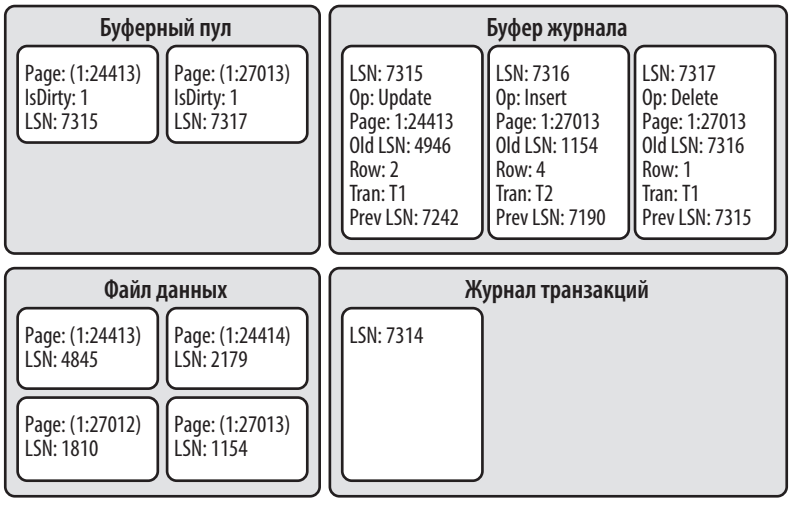


Рис. 11.3. Модификация данных и протоколирование транзакций: состояние после двух модификаций данных

Теперь предположим, что приложение фиксирует транзакцию `T2`. Эта операция создает запись журнала `COMMIT` и заставляет SQL Server записывать (закреплять) содержимое блока журнала на диск. Он переносит *все* записи журнала из буфера на диск независимо от того, какая транзакция их создала (рис. 11.4).

Подтверждение того, что транзакция была зафиксирована, приложения получают *только* после того, как все записи журнала закреплены. Несмотря на то что страница данных (1:27013) все еще «грязная» и не сохранена в файле данных, в закрепленных записях журнала на диске достаточно информации, чтобы, если потребуется, повторно применить модификации, сделанные зафиксированной транзакцией `T2`.

«Грязные» страницы из буферного пула сохраняются в файлы данных на контрольной точке. Эта операция также создает запись журнала `CHECKPOINT` и немедленно закрепляет ее в журнале. Это состояние показано на рис. 11.5.

После контрольной точки страницы в файле данных могут содержать данные из незафиксированных транзакций (`T1` в нашем примере). Однако в записях журнала транзакций достаточно информации, чтобы при необходимости отменить

изменения. В этом случае SQL Server выполняет *компенсационные операции (compensation operations)*: производит действия, обратные к произошедшим изменениям данных, и создает компенсационные записи журнала.

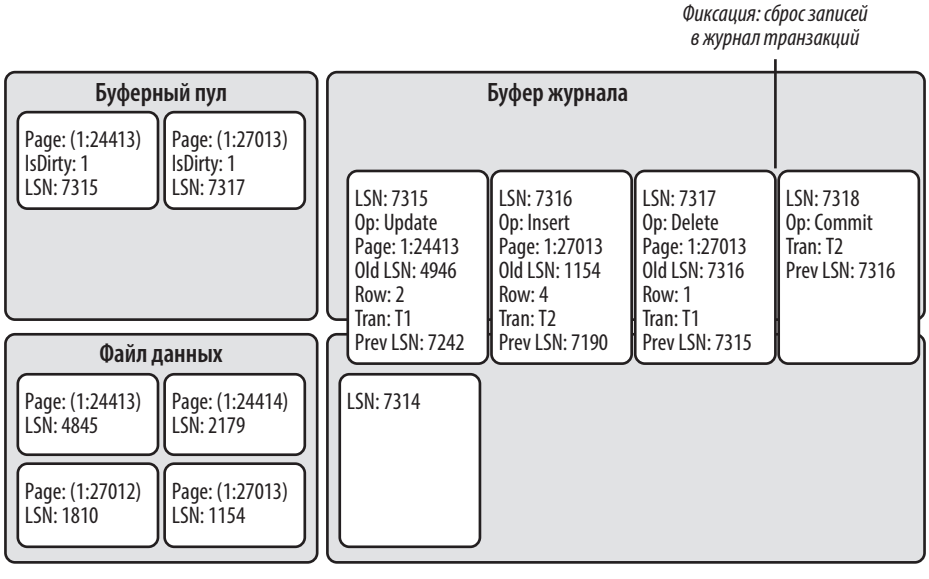


Рис. 11.4. Модификация данных и протоколирование транзакций: операция фиксации

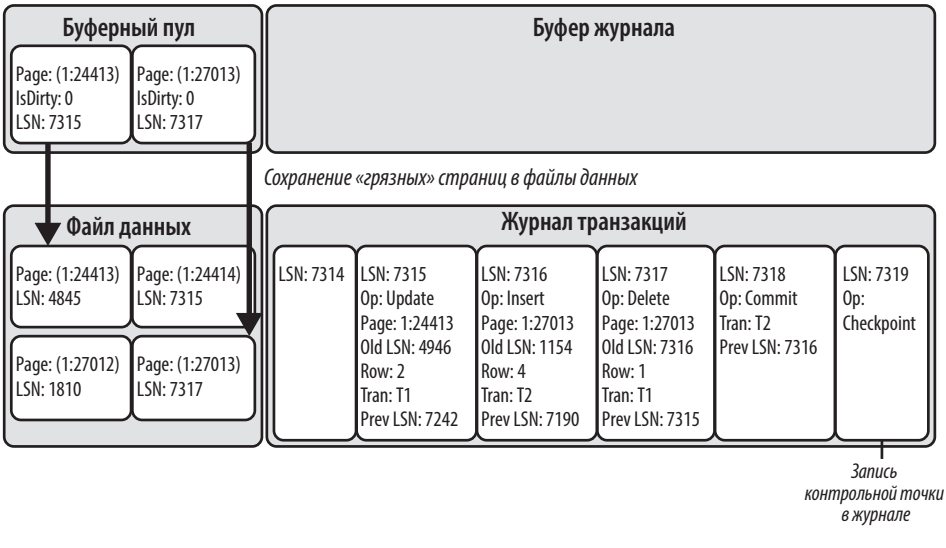


Рис. 11.5. Модификация данных и протоколирование транзакций: контрольная точка

На рис. 11.6 показан пример отката транзакции `T1`. Здесь SQL Server выполнил компенсационное обновление, создав соответствующую запись в журнале (LSN: 7320), чтобы отменить изменения исходной операции обновления (LSN: 7315). Он также выполнил компенсационную вставку (LSN: 7321), чтобы компенсировать операцию удаления (LSN: 7317).

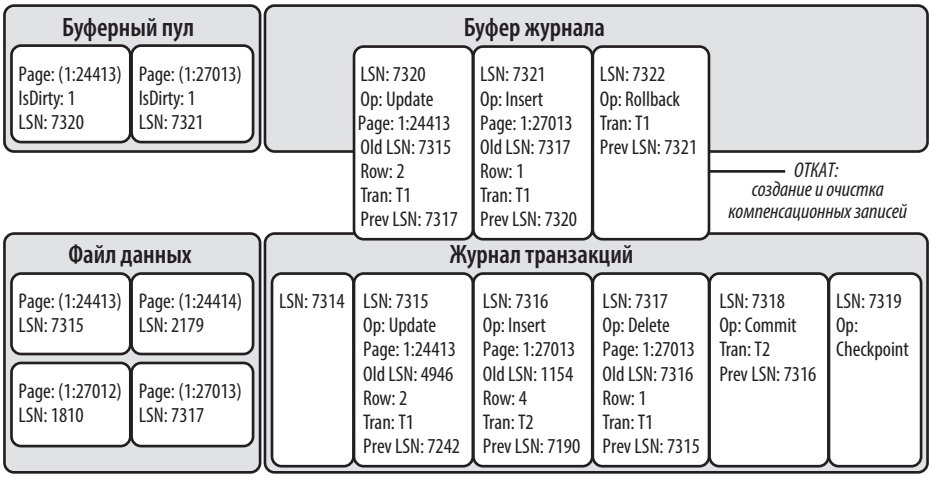


Рис. 11.6. Модификация данных и протоколирование транзакций: откат транзакции

С протоколированием связаны два ожидания, которые стоит отслеживать:

`WRITELOG`

Ожидания `WRITELOG` происходят, когда SQL Server ожидает завершения операции ввода/вывода, которая записывает блок журнала на диск. За исключением отложенной устойчивости (описанной далее в этой главе), этот тип ожидания является синхронным, потому что он не позволяет фиксировать транзакции, пока выполняется ввод/вывод записи. Ваша цель — свести это ожидание к минимуму и повысить пропускную способность журнала транзакций, насколько это возможно.

`LOGBUFFER`

Ожидания `LOGBUFFER` происходят, когда SQL Server ожидает доступного блока журнала для сохранения записей. Чаще всего это случается из-за неудовлетворительной пропускной способности ввода/вывода, когда SQL Server не успевает достаточно быстро записывать блоки журнала на диск. Обычно, когда есть ожидания `LOGBUFFER`, им сопутствуют ожидания `WRITELOG`. Чтобы решить эту проблему, увеличьте пропускную способность журнала транзакций.

Позже в этой главе я расскажу о том, как устранять неполадки и повышать пропускную способность файла журнала.

Чтобы повысить производительность журнала транзакций, также можно уменьшить объемы протоколирования. Для этого можно удалить ненужные и неиспользуемые индексы (подробнее об этом в главе 14), настроить стратегию обслуживания индекса, чтобы сократить разбиения страниц, и уменьшить размер строки в часто модифицируемых индексах.

Можно также улучшить стратегию управления транзакциями, если избегать автоматически фиксируемых транзакций. Это значительно сокращает объем протоколирования и количество запросов ввода/вывода, которые требуются для записи в журнал. Давайте рассмотрим эти вопросы подробнее.

Явные и автоматически фиксируемые транзакции и накладные расходы журнала

Как вы узнали в главе 8, SQL Server всегда выполняет инструкции в контексте транзакции. Если не запущено явных или неявных транзакций, то SQL Server выполняет инструкцию в автоматически фиксируемой транзакции, как если бы инструкция была заключена в блок `BEGIN TRAN ... COMMIT`.

Инструкции управления транзакциями `BEGIN TRAN`, `COMMIT` и `ROLLBACK` создают записи журнала `BEGIN XACT`, `COMMIT XACT` и `ROLLBACK XACT` соответственно. В автоматически фиксируемых транзакциях в записи журнала, протоколирующие каждую инструкцию модификации, также попадают записи `BEGIN XACT` и `COMMIT XACT`. Это может значительно увеличить объем протоколирования и, что более важно, снизить его производительность, потому что SQL Server приходится сбрасывать на диск блоки журнала после каждой инструкции при всех операциях `COMMIT`.

Это показано на рис. 11.7. Операции `INSERT_1`, `UPDATE_1` и `DELETE_1` выполняются в автоматически фиксируемых транзакциях, создавая дополнительные записи в журнале и заставляя буфер журнала очищаться при каждой операции `COMMIT`. Наоборот, если выполнять эти операции в явных транзакциях, протоколирование становится эффективнее.

Код в листинге 11.1 показывает накладные расходы, связанные с автоматически фиксируемыми транзакциями, по сравнению с явными транзакциями. В обоих режимах он выполняет последовательность `INSERT`/`UPDATE`/`DELETE` по 10 000 раз в цикле, а затем сравнивает время выполнения инструкций и пропускную способность журнала транзакций с помощью представления `sys.dm_io_virtual_file_stats`.

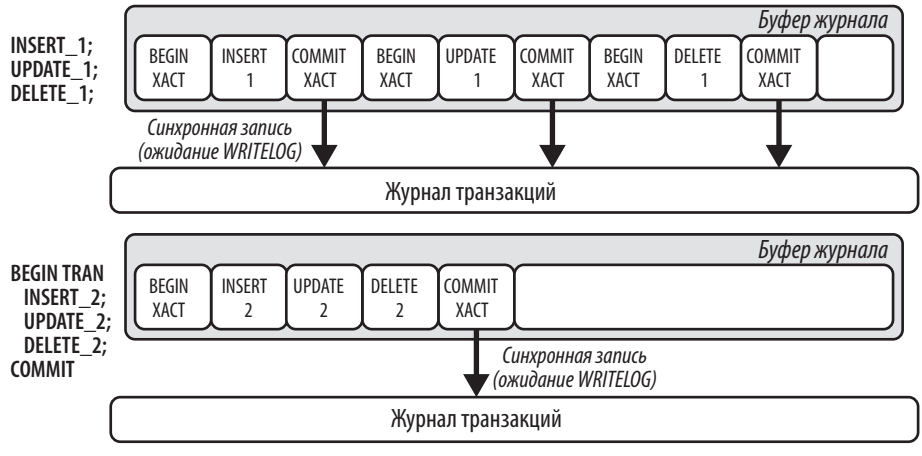


Рис. 11.7. Явные и автоматически фиксируемые транзакции

Листинг 11.1. Явные и автоматически фиксируемые транзакции

```
CREATE TABLE dbo.TranOverhead
(
    Id INT NOT NULL,
    Col CHAR(50) NULL,
    CONSTRAINT PK_TranOverhead
    PRIMARY KEY CLUSTERED(Id)
);

-- Автоматически фиксируемые транзакции
DECLARE
    @Id INT = 1
    ,@StartTime DATETIME = GETDATE()
    ,@num_of_writes BIGINT
    ,@num_of_bytes_written BIGINT

SELECT @num_of_writes = num_of_writes, @num_of_bytes_written =
                        num_of_bytes_written
FROM sys.dm_io_virtual_file_stats(db_id(),2);

WHILE @Id <= 10000
BEGIN
    INSERT INTO dbo.TranOverhead(Id, Col) VALUES(@Id, 'A');
    UPDATE dbo.TranOverhead SET Col = 'B' WHERE Id = @Id;
    DELETE FROM dbo.TranOverhead WHERE Id = @Id;

    SET @Id += 1;
END;

SELECT
    DATEDIFF(MILLISECOND,@StartTime,GETDATE())
```

```
        AS [Time(ms): Autocommitted Tran]
    ,s.num_of_writes - @num_of_writes
        AS [Number of writes]
    ,(s.num_of_bytes_written - @num_of_bytes_written) / 1024
        AS [Bytes written (KB)]
FROM
    sys.dm_io_virtual_file_stats(db_id(),2) s;
GO

-- Явные транзакции
DECLARE
    @Id INT = 1
    ,@StartTime DATETIME = GETDATE()
    ,@num_of_writes BIGINT
    ,@num_of_bytes_written BIGINT

SELECT @num_of_writes = num_of_writes, @num_of_bytes_written =
                        num_of_bytes_written
FROM sys.dm_io_virtual_file_stats(db_id(),2);

WHILE @Id <= 10000
BEGIN
    BEGIN TRAN
        INSERT INTO dbo.TranOverhead(Id, Col) VALUES(@Id, 'A');
        UPDATE dbo.TranOverhead SET Col = 'B' WHERE Id = @Id;
        DELETE FROM dbo.TranOverhead WHERE Id = @Id;
    COMMIT
    SET @Id += 1;
END;

SELECT
    DATEDIFF(MILLISECOND,@StartTime,GETDATE())
        AS [Time(ms): Explicit Tran]
    ,s.num_of_writes - @num_of_writes
        AS [Number of writes]
    ,(s.num_of_bytes_written - @num_of_bytes_written) / 1024
        AS [Bytes written (KB)]
FROM
    sys.dm_io_virtual_file_stats(db_id(),2) s;
```

Вывод кода в моей среде показан на рис. 11.8. Явные транзакции оказались примерно втрое быстрее и создавали втрое меньше протоколирования, чем автоматически фиксируемые.

	Time(ms): Autocommitted Tran	Number of writes	Bytes written (KB)
1	13117	30000	15670
	Time(ms): Explicit Tran	**Number of writes**	**Bytes written (KB)**
1	4500	10000	10658

Рис. 11.8. Производительность явных и автоматически фиксируемых транзакций

Как я уже говорил, грамотное использование явных транзакций может значительно повысить пропускную способность журнала транзакций. Однако не забывайте о том, как длительные транзакции влияют на блокирование. Монопольные блокировки (X) удерживаются до конца транзакции. При разработке стратегии транзакций и при написании кода учитывайте поведение блокировок, а также факторы, рассмотренные в главе 8.

К сожалению, в существующих системах не всегда есть возможность изменить стратегию транзакций. Если ваша система страдает от большого количества автоматически фиксируемых транзакций и может допустить небольшую потерю данных, попробуйте другую функцию — *отложенную устойчивость (delayed durability)*. Она доступна в SQL Server 2014 и более поздних версиях.

Отложенная устойчивость

Как вы уже знаете, SQL Server сбрасывает содержимое блока журнала в файл журнала в момент фиксации транзакции. SQL Server направляет подтверждение клиенту только после того, как запись фиксации закреплена на диске. В случае автоматически фиксируемых транзакций это может привести ко множеству небольших запросов ввода/вывода для записи в журнал.

Отложенная устойчивость изменяет это поведение и делает операции фиксации асинхронными. Клиент немедленно получает подтверждение того, что транзакция была зафиксирована, не дожидаясь, пока запись фиксации будет закреплена на диске. Запись фиксации остается в буфере журнала до тех пор, пока не наступит одно или несколько из следующих условий.

- Блок журнала заполнен.
- В той же базе данных зафиксирована полностью устойчивая транзакция, и ее запись фиксации сбрасывает содержимое буфера журнала на диск.
- Выполняется операция `CHECKPOINT`.
- Вызвана хранимая процедура `sp_flush_log`.
- Запущена операция очистки буфера журнала на основании скорости протоколирования и/или достижения порогового времени ожидания.

Это создает определенный риск. Если до того, как запись фиксации закреплена, на SQL Server произойдет сбой, то изменения данных транзакции с отложенной устойчивостью откатятся при восстановлении, как будто транзакция вообще никогда не фиксировалась. Однако в промежутке времени между фиксацией транзакции и сбоем другие транзакции смогут увидеть изменения данных, сделанные этой транзакцией.

Отложенную устойчивость можно контролировать как на уровне базы данных, так и на уровне транзакций. Параметр базы данных `DELAYED_DURABILITY` поддерживает три различных значения:

`DISABLED`

Это значение по умолчанию отключает отложенную устойчивость во всей базе данных независимо от режима устойчивости отдельных транзакций. Все транзакции в базе всегда полностью устойчивы.

`FORCED`

Это значение обеспечивает отложенную устойчивость для всех транзакций базы данных независимо от режима устойчивости отдельных транзакций.

`ALLOWED`

В этом случае отложенная устойчивость контролируется на уровне транзакций. Все транзакции, для которых она не задана, являются полностью устойчивыми. В листинге 11.2 показано, как задать режим устойчивости на уровне транзакции.

Листинг 11.2. Управление отложенной устойчивостью на уровне транзакции

```
BEGIN TRAN
/* Выполняем операции */
COMMIT WITH (DELAYED_DURABILITY=ON);
```

Отложенную устойчивость можно использовать в оживленных системах с большим количеством автоматически фиксируемых транзакций и недостаточной пропускной способностью журналов. Но чаще всего я предпочитаю обходиться без нее. Я прибегаю к этой функции лишь в крайнем случае, когда не удается повысить пропускную способность журналов другими методами и *только* если допустимы некоторые потери данных. Используйте отложенную устойчивость с особой осторожностью!

Протоколирование транзакций In-Memory OLTP

Хотя подробное описание In-Memory OLTP в памяти выходит за рамки этой книги, стоит упомянуть о том, как эта технология ведет журнал транзакций. В отличие от технологий, основанных на строках и столбцах, In-Memory OLTP создает записи журнала транзакций в момент операции `InCOMMIT`, причем только если транзакция успешно зафиксирована. Протоколирование тоже оптимизировано. Каждая транзакция обычно создает всего одну или несколько записей журнала транзакций, даже если она изменяет большие объемы данных. Эти записи хранятся в обычном файле журнала и включаются в резервные копии наряду со всеми остальными записями журнала.

Такое поведение может изменить характер ввода/вывода для операций журнала. При записи в журнал In-Memory OLTP может создавать более объемные запросы на запись, особенно при крупных транзакциях In-Memory OLTP. Кроме того, файлы журналов постоянно считываются непрерывным процессом контрольной точки, который анализирует записи журнала и обновляет данные In-Memory OLTP, сохраняемые на диске.

В большинстве случаев об этих подробностях не нужно беспокоиться. Достаточно помнить об особом характере ввода/вывода, если вы разрабатываете подсистему ввода/вывода для баз данных, использующих In-Memory OLTP.

Виртуальные файлы журналов

На внутреннем уровне SQL Server делит физические файлы журналов на меньшие части, называемые *виртуальными файлами журналов* (*VLF, virtual log files*). SQL Server использует их как единицы управления, и они могут быть активными или неактивными.

Активные VLF хранят активную часть журнала транзакций, которая содержит записи журнала, необходимые, чтобы обеспечивать транзакционную согласованность базы данных, восстановление к определенному моменту времени и активные процессы SQL Server, которые опираются на записи журнала (например, репликацию транзакций и группы доступности AlwaysOn). Неактивные VLF содержат усеченные (неактивные) и неиспользуемые части журнала транзакций.

На рис. 11.9 показан пример файла журнала транзакций и соответствующие VLF. Активная часть журнала начинается с VLF3, где протоколируется самая старая из активных транзакций в системе. В случае отката SQL Server потребуется доступ к записям журнала, созданным этой транзакцией.

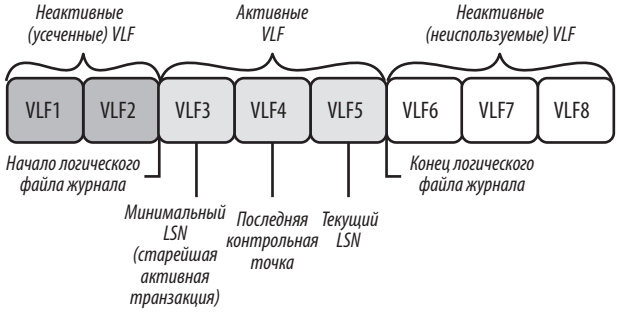


Рис. 11.9. Журнал транзакций и VLF

На рис. 11.9 единственный процесс, из-за которого VLF3 остается активным, — это активная транзакция. Когда она фиксируется, SQL Server усекает журнал,

помечая VLF3 как неактивный (рис. 11.10). Усечение журнала транзакций не уменьшает размер файла журнала на диске, а лишь означает, что части журнала транзакций (один или несколько VLF) помечены как неактивные и доступны для повторного использования. (Этот пример намеренно упрощен; вскоре я подробнее расскажу о принципах усечения журнала.)

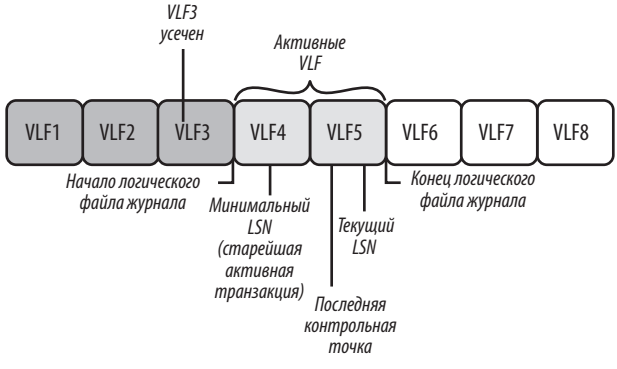


Рис. 11.10. Журнал транзакций и VLF после фиксации

SQL Server использует VLF в качестве единицы усечения. VLF не может быть помечен как неактивный, если содержит хотя бы одну запись из активной части журнала. Это одна из причин, по которой VLF не должны быть очень большими.

Механизм усечения журналов зависит от выбранной *модели восстановления базы данных*. Существуют три модели восстановления, каждая из которых гарантирует, что в активной части журнала будет достаточно данных, чтобы обеспечить согласованность базы данных. Но разные модели предусматривают разные уровни восстановимости в случае сбоя, а при моделях `SIMPLE` и `BULK-LOGGED` некоторые технологии SQL Server будут недоступны.

Давайте подробнее рассмотрим модели восстановления.

`SIMPLE`

В модели восстановления `SIMPLE` журнал усекается в контрольной точке. На диск сохраняются все страницы данных, чей LSN меньше, чем у контрольной точки. В случае неожиданного завершения работы или сбоя SQL Server не требуется обращаться к записям журнала, предшествующим контрольной точке, чтобы повторно применить их к страницам данных. Из-за старых активных транзакций и репликации транзакций усечение журнала может откладываться и VLF будут оставаться активными.

В этой модели восстановления SQL Server не использует резервные копии журналов транзакций. Поэтому нельзя выполнить восстановление на

определенный момент времени и можно потерять данные, если поврежден какой-либо из файлов базы данных (файл данных или файл журнала). *Точка восстановления* (RPO, recovery point) базы данных в этой модели становится моментом последней полной резервной копии.

В некоторых случаях модель восстановления `SIMPLE` может быть вполне приемлемой: например, когда данные статичны или когда их можно воссоздать из других источников. Однако если вы сталкиваетесь с этой моделью при проверке работоспособности системы, стоит обсудить с заинтересованными сторонами, насколько она приемлема и допустима ли потеря данных.

`FULL`

В модели восстановления `FULL` SQL Server полностью протоколирует все операции, выполняющиеся в базе данных, и требует регулярного резервного копирования журнала транзакций, чтобы его можно было безопасно усекать. Поскольку в резервных копиях журнала транзакций хранятся все записи журнала в базе данных, этот режим поддерживает восстановление на определенный момент времени, если доступна последовательность файлов резервных копий (*цепочка резервных копий*). Модель `FULL` обычно стоит использовать в промышленных базах данных за некоторыми исключениями.

Модель восстановления `FULL` требуется для доступа к различным функциям и технологиям SQL Server, которые опираются на записи журнала транзакций. Эти технологии также могут откладывать усечение журнала, даже если резервные копии уже созданы. Например, если узел группы доступности AlwaysOn отключается, то SQL Server не сможет усечь журнал до тех пор, пока узел не подключится снова и не «догонит» вторичную реплику.

`BULK-LOGGED`

Модель восстановления `BULK-LOGGED` работает так же, как `FULL`, за исключением того, что для некоторых операций применяется минимальное протоколирование: например, для создания индекса или инструкций `BULK INSERT`. При минимальном протоколировании SQL Server регистрирует в файле журнала только информацию о распределении экстентов. При этом файл журнала используется меньше, но из-за минимального протоколирования становится невозможным восстановление базы данных на определенный момент времени. Если в промышленной среде вы встречаете базы данных с моделью восстановления `BULK-LOGGED`, то, как и в случае `SIMPLE`, стоит проанализировать частоту операций с неполным протоколированием и риски потенциальной потери данных.

Анализировать VLF можно с помощью представления `sys.dm_db_log_info`[1] в SQL Server 2016 и более поздних версиях или с помощью команды `DBCC LOGINFO` в более ранних версиях. В листинге 11.3 показан код, где используется это представление для одной из баз данных.

[1] https://oreil.ly/oZ7DS

Листинг 11.3. Анализ VLF в базе данных

```
SELECT *
FROM sys.dm_db_log_info(DB_ID());

SELECT
    COUNT(*) as [VLF Count]
    ,MIN(vlf_size_mb) as [Min VLF Size (MB)]
    ,MAX(vlf_size_mb) as [Max VLF Size (MB)]
    ,AVG(vlf_size_mb) as [Avg VLF Size (MB)]
FROM sys.dm_db_log_info(DB_ID());
```

На рис. 11.11 показаны выходные данные этого представления для базы данных с неправильной конфигурацией файла журнала, где автоувеличение настроено на величину 10 %. Как видите, разные VLF в базе данных сильно отличаются друг от друга по объему.

	database_id	file_id	vlf_begin_offset	vlf_size_mb	vlf_sequence_number	vlf_active	vlf_s
35	16	2	27459584	2.62	281	1	2
36	16	2	30212096	2.87	282	1	2
37	16	2	33226752	3.18	283	1	2
38	16	2	36569088	3.5	284	1	2
39	16	2	40239104	3.87	285	1	2
40	16	2	44302336	4.25	286	1	2
41	16	2	48758784	4.68	287	1	2
42	16	2	53673984	5.12	288	1	2
43	16	2	59047936	5.62	289	1	2

	VLF Count	Min VLF Size (MB)	Max VLF Size (MB)	Avg VLF Size (MB)
1	105	0.24	1889.87	197.98695238095235

Рис. 11.11. Неэффективная конфигурация VLF

Конфигурация журнала транзакций

SQL Server работает с журналами транзакций последовательно, записывая и читая поток записей журнала. Хотя журнал может состоять из нескольких физических файлов, это не дает особых преимуществ: чаще всего один файл журнала удобнее поддерживать и контролировать.

Существует несколько особых случаев, когда полезно иметь несколько файлов журналов:

- SQL Server может параллельно инициализировать файл журнала нулями. Это может ускорить создание или восстановление базы данных, если

в ней используются большие (порядка нескольких терабайт) файлы журналов.

- Если вы хотите разместить журнал транзакций на быстром, но небольшом диске, можно создать файл, предварительно выделив под него все дисковое пространство, и добавить еще один небольшой файл журнала на более вместительном и медленном диске. Бóльшую часть времени SQL Server будет работать с файлом на быстром диске, а второй небольшой файл пригодится, если окажется, что журнал транзакций заполнен и не усекается.

Лучше управлять размером журнала транзакций вручную, избегая накладных расходов на инициализацию нулями во время автоувеличения. Можно проанализировать размер журнала транзакций и заново создать журнал, предварительно выделив нужный объем. При анализе обязательно учитывайте операции с интенсивным использованием журналов, такие как обслуживание индекса.

Из главы 1 вы знаете, как можно перестроить журнал: сократить его до минимального размера, а затем предварительно выделить пространство, используя фрагменты объемом от 1024 до 4096 Мбайт. Я обычно применяю фрагменты по 1024 Мбайт, в результате чего создаются VLF размером по 128 Мбайт. Если мне нужны *очень* большие файлы журналов — размером в сотни гигабайт или даже несколько терабайт, — я могу использовать более крупные фрагменты.

Не ограничивайте максимальный размер журнала и его автоматическое увеличение: у журнала должна быть возможность увеличиваться в случае чрезвычайных ситуаций!

Проблемы с усечением журнала

Чрезмерный рост журнала транзакций — распространенная проблема, с которой стоит уметь справляться даже неопытным администраторам баз данных. Такой рост происходит, когда SQL Server не может усечь журнал транзакций и повторно использовать его пространство. В этих случаях файл журнала продолжает увеличиваться, пока не заполнит весь диск. База данных переключается в режим только для чтения, и появляется ошибка «Transaction log full» («Журнал транзакций заполнен») (код ошибки 9002).

Лучший способ бороться с этой ситуацией — не попадать в нее. Как я говорил в главе 9, важно следить за свободным местом на диске и настраивать уведомления. Кроме того, можно предварительно выделить под файл журнала все пространство на диске, отслеживать объем свободного места в файле журнала и настроить уведомления о том, что место заканчивается.

Если случилось, что «Журнал транзакций заполнен», самое главное — не паникуйте. Прежде всего изучите первопричину проблемы и выясните, получится ли быстро все исправить. Для этого можно просмотреть столбец `log_reuse_wait_desc` в представлении `sys.databases`, либо запросив его напрямую, либо используя более сложный код, показанный в листинге 11.4. В этом столбце показано, почему журнал не усекается.

Листинг 11.4. Анализ столбца `log_reuse_wait_desc` в представлении `sys.databases`

```
CREATE TABLE #SpaceUsed
(
    database_id SMALLINT NOT NULL,
    file_id SMALLINT NOT NULL,
    space_used DECIMAL(15,3) NOT NULL,
    PRIMARY KEY(database_id, file_id)
);

EXEC master..sp_MSforeachdb
N'USE[?];
INSERT INTO #SpaceUsed(database_id, file_id, space_used)
    SELECT DB_ID(''?''), file_id,
        (size – CONVERT(INT,FILEPROPERTY(name, ''SpaceUsed''))) / 128.
FROM sys.database_files
WHERE type = 1;';

SELECT
    d.database_id, d.name, d.recovery_model_desc
    ,d.state_desc, d.log_reuse_wait_desc, m.physical_name
    ,m.is_percent_growth
    ,IIF(m.is_percent_growth = 1
        ,m.growth
        ,CONVERT(DECIMAL(15,3),m.growth / 128.0)
    ) AS [Growth (MB or %)]
    ,CONVERT(DECIMAL(15,3),m.size / 128.0) AS [Size (MB)]
    ,IIF(m.max_size = -1
        ,-1
        ,CONVERT(DECIMAL(15,3),m.max_size / 128.0)
    ) AS [Max Size(MB)]
    ,s.space_used as [Space Used(MB)]
FROM
    sys.databases d WITH (NOLOCK)
        JOIN sys.master_files m WITH (NOLOCK) ON
        d.database_id = m.database_id
    LEFT OUTER JOIN #SpaceUsed s ON
        s.database_id = m.database_id AND
        s.file_id = m.file_id
ORDER BY
    d.database_id;
```

На рис. 11.12 показан результат работы листинга 11.4.

Рассмотрим наиболее распространенные причины, по которым усечение журнала откладывается, и соответствующие значения в `log_reuse_wait_desc`.

database…	name	recovery…	state_desc	log_reuse_wait_desc	physical_…	is_percent_growth	Growth (MB or %)	Size (MB)	Max Size(MB)	Space Used(MB)
1	master	SIMPLE	ONLINE	NOTHING	M:\SQLDa…	1	10.000	2.250	-1.000	1.328
2	tempdb	SIMPLE	ONLINE	NOTHING	T:\SQLDa…	0	1024.000	1024.000	-1.000	966.094
2	tempdb	SIMPLE	ONLINE	NOTHING	L:\SQLDa…	0	1024.000	1024.000	2097152.000	966.094
3	model	FULL	ONLINE	LOG_BACKUP	M:\SQLDa…	0	1024.000	1024.000	-1.000	1015.336
4	msdb	SIMPLE	ONLINE	NOTHING	M:\SQLDa…	1	10.000	19.625	2097152.000	18.133
5	DBA	SIMPLE	ONLINE	NOTHING	L:\SQLDa…	0	1024.000	1024.000	2097152.000	937.281
16	LogDemo2	FULL	ONLINE	LOG_BACKUP	L:\SQLDa…	0	0.000	20769.0…	21000.000	-16.281

Рис. 11.12. Анализ данных `log_reuse_wait_desc`

Ожидание повторного использования журнала LOG_BACKUP

Ожидание повторного использования журнала `LOG_BACKUP` — одно из наиболее распространенных ожиданий в базах данных с моделями восстановления `FULL` и `BULK-LOGGED`. Оно обозначает, что журнал не может быть усечен из-за отсутствия последних резервных копий журнала транзакций.

Встретив это ожидание, проверьте состояние задания резервного копирования журнала транзакций. Убедитесь, что работа задания не нарушена из-за нехватки пространства в месте назначения резервного копирования или по другим причинам. Также возможно, что во время операций с интенсивным использованием журналов частота резервного копирования журнала недостаточно высока. Например, обслуживание индекса может создать огромное количество записей в журнале транзакций за очень короткое время.

Эту проблему можно смягчить, выполнив резервное копирование журнала транзакций. Не забудьте сохранить файл резервной копии, если вы выполняете эту операцию вручную и задаете нестандартное место назначения резервного копирования. Файл станет частью цепочки резервных копий и понадобится для восстановления базы данных.

Если вы не используете технологии, опирающиеся на модель восстановления `FULL`, то можно временно переключить базу данных в режим `SIMPLE`, отчего журнал транзакций будет усечен. Помните, что это может привести к потере данных. Как можно скорее вернитесь к модели восстановления `FULL` и немедленно повторно инициализируйте цепочку резервного копирования, выполнив операции резервного копирования `FULL` и `LOG`.

Наконец, этой ситуации часто можно избежать, если правильно отслеживать работоспособность заданий резервного копирования. Настройте уведомления о повторяющихся сбоях резервного копирования журнала.

Ожидание повторного использования журнала ACTIVE_TRANSACTION

Ожидание повторного использования журнала `ACTIVE_TRANSACTION` указывает на то, что журнал не удается усечь из-за старой незавершенной транзакции. Особенно часто это происходит из-за того, что в приложении неправильно настроено управление транзакциями, так что оно порождает неуправляемые незафиксированные транзакции. Например, приложение может запустить несколько инструкций `BEGIN TRAN` без соответствующих инструкций `COMMIT`.

Будьте осторожны, если запускаете явные или неявные транзакции в SQL Server Management Studio (SSMS). Если не включена настройка `SET XACT_ABORT ON`, SSMS не будет откатывать транзакцию, когда вы принудительно завершаете пакетное выполнение.

Список активных транзакций можно посмотреть с помощью кода из листинга 11.5. Код может вывести несколько строк для каждой транзакции, потому что получает информацию об использовании журнала для каждой базы данных по отдельности.

Листинг 11.5. Получение списка активных транзакций

```
SELECT
    dt.database_id
    ,DB_NAME(dt.database_id) as [DB]
    ,st.session_id
    ,CASE at.transaction_state
        WHEN 0 THEN 'Not Initialized'
        WHEN 1 THEN 'Not Started'
        WHEN 2 THEN 'Active'
        WHEN 3 THEN 'Ended (R/O)'
        WHEN 4 THEN 'Commit Initialize'
        WHEN 5 THEN 'Prepared'
        WHEN 6 THEN 'Committed'
        WHEN 7 THEN 'Rolling Back'
        WHEN 8 THEN 'Rolled Back'
    END AS [State]
    ,at.transaction_begin_time
    ,es.login_name
    ,ec.client_net_address
    ,ec.connect_time
    ,dt.database_transaction_log_bytes_used
    ,dt.database_transaction_log_bytes_reserved
    ,er.status
    ,er.wait_type
    ,er.last_wait_type
    ,sql.text AS [SQL]
FROM
    sys.dm_tran_database_transactions dt WITH (NOLOCK)
```

```
        JOIN sys.dm_tran_session_transactions st WITH (NOLOCK) ON
            dt.transaction_id = st.transaction_id
        JOIN sys.dm_tran_active_transactions at WITH (NOLOCK) ON
            dt.transaction_id = at.transaction_id
        JOIN sys.dm_exec_sessions es WITH (NOLOCK) ON
            st.session_id = es.session_id
        JOIN sys.dm_exec_connections ec WITH (NOLOCK) ON
            st.session_id = ec.session_id
        LEFT OUTER JOIN sys.dm_exec_requests er WITH (NOLOCK) ON
            st.session_id = er.session_id
        CROSS APPLY
            sys.dm_exec_sql_text(ec.most_recent_sql_handle) sql
ORDER BY
    dt.database_transaction_begin_time;
```

Сеанс, удерживающий активную транзакцию, можно принудительно завершить с помощью команды `KILL`. Позже можно проанализировать, почему транзакция не была обработана корректно.

Ожидание повторного использования журнала AVAILABILITY_REPLICA

По названию нетрудно догадаться, что ожидание повторного использования журнала `AVAILABILITY_REPLICA` возникает в системах, использующих группы доступности `AlwaysOn`. В этой технологии первичный узел взаимодействует со вторичными узлами, отправляя им поток записей журнала транзакций. Журнал не усекается до тех пор, пока эти записи не будут отправлены и воспроизведены на вторичных репликах.

Ожидания `AVAILABILITY_REPLICA` обычно возникают при определенных проблемах в группах доступности: например, когда вторичный узел недоступен, репликация между узлами запаздывает или вторичные реплики не успевают воспроизводить изменения со скоростью нагрузки.

Если встречается это ожидание, проверьте работоспособность группы доступности. В большинстве случаев единственный быстрый способ устранения неполадок, помимо увеличения места в журнале, — удалить проблемный вторичный узел из группы доступности. Подробнее о группах доступности я расскажу в главе 12.

Ожидание повторного использования журнала DATABASE_MIRRORING

Ожидание повторного использования журнала `DATABASE_MIRRORING` возникает в системах, где используется технология зеркального отображения баз данных.

Она предшествовала группам доступности AlwaysOn и ведет себя похожим образом, взаимодействуя через поток записей журнала.

Мы не будем затрагивать устранение неполадок зеркального отображения, потому что эта технология давно устарела. Концептуально неполадки устраняются примерно так же, как в группах доступности.

Увидев ожидание `DATABASE_MIRRORING`, как и в случае с `AVAILABILITY_REPLICA`, проанализируйте работоспособность зеркального отображения базы данных.

Ожидание повторного использования журнала REPLICATION

Ожидание повторного использования журнала `REPLICATION` возникает, когда агент чтения журнала запаздывает, собирая записи журнала для процессов репликации транзакций или отслеживания измененных данных (CDC). Увидев это ожидание, проверьте состояние агента чтения журнала и устраните обнаруженные проблемы.

Одна из возможных проблем заключается в том, что у запроса агента чтения журнала (Log Reader Agent) истекло время ожидания. По умолчанию оно равно 30 минутам, чего в большинстве случаев достаточно. Однако этого времени может не хватить, если система обрабатывает очень большие модификации данных (миллионы строк) в реплицированных таблицах. В этом случае можно увеличить параметр `QueryTimeout` в профиле агента чтения журнала. Подробности можно найти в документации Microsoft[1].

Есть два железобетонных варианта: можно удалить репликацию, а можно пометить записи журнала как собранные с помощью команды `sp_repldone`. В обоих случаях впоследствии может потребоваться повторно инициалировать репликацию.

Ожидание повторного использования журнала ACTIVE_BACKUP_OR_RESTORE

Ожидание повторного использования журнала `ACTIVE_BACKUP_OR_RESTORE` указывает на то, что журнал не усекается из-за активных процессов резервного копирования или восстановления базы данных независимо от того, какой тип резервного копирования или восстановления выполняется.

Это ожидание часто возникает из-за того, что снижается производительность сети или хранилища резервных копий во время резервного копирования в ре-

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

жиме FULL. Обнаружив это ожидание, проверьте состояние активных заданий резервного копирования и восстановления.

Другие стратегии решения проблем с усечением журнала

Не всегда удается быстро устранить первопричину проблемы, а иногда это и вовсе невозможно. Например, если удалить недоступную реплику из группы доступности или отключить репликацию, то впоследствии может понадобиться значительный объем работы, чтобы создать их повторно.

В качестве временного решения можно добавить еще один файл журнала или увеличить объем диска. Тогда база данных продолжит работать, а у вас появится время на то, чтобы устранить основную причину проблемы.

Имейте в виду, что усечению журнала транзакций могут препятствовать несколько проблем одновременно. Например, сбой в сети может помешать серверу не только взаимодействовать с репликами группы доступности, но и получить доступ к месту назначения резервного копирования. После устранения каждой проблемы проверяйте значение log_reuse_wait_desc и объем свободного места в файле журнала.

Наконец, учитесь на собственном опыте. Проблемы вида «Журнал транзакций заполнен» — это серьезные неприятности, и их следует избегать любой ценой. Изучите первопричины, проанализируйте свою стратегию мониторинга и переосмыслите планирование мощностей, чтобы уменьшить вероятность повторения проблемы.

Ускоренное восстановление базы данных

Ускоренное восстановление базы данных (ADR, accelerated database recovery) — это новая функция, появившаяся в SQL Server 2019. Она может значительно сократить время восстановления базы данных, если SQL Server откажет посреди длительной транзакции. Еще одно преимущество ADR — в том, что оно позволяет SQL Server усекать журнал транзакций при наличии длительных активных транзакций, что уменьшает или даже устраняет ожидания повторного использования журнала ACTIVE_TRANSACTION.

К сожалению, эта функция добавляет накладные расходы. На внутреннем уровне она использует хранилище версий, концептуально похожее на традиционное хранилище версий в tempdb. Дополнительная нагрузка, связанная с этим хранилищем, может оказаться заметной, а может и нет — все зависит от рабочей нагрузки системы и аппаратного обеспечения.

Отдельно подчеркну, что ADR решает очень специфическую проблему, а именно медленное восстановление базы данных, связанное с наличием длительных

активных транзакций. Эта функция не помогает в других случаях — например, если вы аварийно переключаете группу доступности на вторичный узел с большой очередью повторного выполнения (подробнее об этом в главе 12). Из-за накладных расходов, связанных с ADR, не рекомендуется включать эту функцию, если она не нужна в вашей системе.

На мой взгляд, ADR — палка о двух концах. Оно помогает замаскировать проблемы длительных активных транзакций, но не устраняет основную причину проблемы. В некоторых случаях такие транзакции могут быть уместны, однако всегда лучше перепроектировать систему так, чтобы избегать их, насколько это возможно.

ADR включено по умолчанию в базах данных и управляемых экземплярах Azure SQL. Эту функцию можно включить и в обычном SQL Server с помощью параметра базы данных `ALTER DATABASE SET ACCELERATED_DATABASE_RECOVERY`. Следить за размером постоянного хранилища версий (PVS) в базе данных можно с помощью представления `sys.dm_tran_persistent_version_store_stats`. Подробнее об этой функции можно прочесть в документации Microsoft[1].

Пропускная способность журнала транзакций

Влияние недостаточной пропускной способности журнала транзакций не всегда заметно. Хотя любая операция, которая изменяет что-то в базе данных, также делает запись в журнале транзакций, эти записи считаются частью операции. Специалисты часто смотрят на общую картину и оценивают производительность всей операции в целом, упуская из виду влияние отдельных компонентов.

Рассмотрим, например, обслуживание индекса. Эта операция чрезвычайно интенсивно обращается к журналам, и плохая пропускная способность журналов сильно влияет на ее производительность. Однако когда администраторы баз данных пытаются облегчить обслуживание индексов, они обычно корректируют его расписание или исключают индексы из процесса обслуживания, но игнорируют при этом медленную запись в журнал. (Исключения бывают, но они редки и немногочисленны.)

Медленная запись в журнал также часто остается незамеченной на фоне обычной рабочей нагрузки в системах OLTP. Из-за высокой задержки записи в журнал увеличивается время выполнения запросов, однако на это редко обращают внимание при настройке запросов. В любом случае от повышения производительности журнала транзакций всегда повышается и производительность системы.

---

[1] https://oreil.ly/79pCX

Тем не менее хочу предостеречь: хотя повышение производительности журнала транзакций всегда полезно, это не волшебная палочка для решения всех проблем, и его эффект не всегда будет заметен пользователям. Возможно, ваши усилия больше окупятся, если сначала устранить другие узкие места.

Анализируя статистику ожидания, можно посмотреть на ожидания `WRITELOG` и `LOGBUFFER`. Как я уже упоминал, ожидание `WRITELOG` возникает, когда SQL Server ожидает завершения записи в журнал. Ожидание `LOGBUFFER` возникает, когда у SQL Server нет доступного буфера журнала, чтобы кэшировать записи журнала, или когда он не может достаточно быстро сбросить блоки журнала на диск.

Я бы с радостью указал точный процентный порог, когда эти ожидания становятся проблемой, но это невозможно. Так или иначе они всегда будут в системе, но чтобы оценить, насколько они критичны, стоит проанализировать работоспособность ввода/вывода и его пропускную способность. Ожидания `WRITELOG` часто наблюдаются вместе с другими ожиданиями ввода/вывода (например, `PAGEIOLATCH`), особенно когда файлы журнала и данных используют одно и то же *физическое* хранилище и сетевой путь ввода/вывода. В большинстве случаев это признак того, что подсистема ввода/вывода либо перегружена, либо неправильно настроена.

Как упоминалось в главе 3, представление `sys.dm_io_virtual_file_stats` содержит информацию о задержках и пропускной способности файла базы данных. Если используется сетевое хранилище, меня обычно устраивает средняя задержка записи в файлы журнала в пределах 1–2 мс.

В высокопроизводительных системах OLTP файлы журналов можно размещать в хранилище с прямым подключением (DAS) на накопителях NMVe, что должно снизить задержку до значений меньше миллисекунды. В крайнем случае можно использовать технологии постоянной памяти, чтобы задержка стала еще меньше.

Обращайте внимание на средний размер записи в журнал. Небольшие записи менее эффективны и могут ухудшить пропускную способность журнала. В большинстве случаев такие записи — признак автоматически фиксируемых транзакций. Их количество можно уменьшить, если грамотно управлять транзакциями или включить отложенную устойчивость (если вы готовы к небольшой потере данных).

В объекте `Databases` есть несколько счетчиков производительности, с помощью которых можно отслеживать активность журнала транзакций в режиме реального времени. Вот эти счетчики:

`Log Bytes Flushed/sec`

Количество байт данных, записанных в файл журнала в секунду.

`Log Flushed/sec`

Количество операций записи в журнал в секунду. Этот счетчик можно использовать вместе с `Log Bytes Flush/sec`, чтобы оценить средний размер записи журнала.

`Log Flush Write Time(ms)`

Среднее время операции записи в журнал. С помощью этого счетчика можно увидеть задержку ввода/вывода при записи в журнал в режиме реального времени.

К сожалению, счетчик измеряет время в целых миллисекундах и показывает 0, если подсистема хранения дает задержку менее миллисекунды.

`Log Flush Waits/sec`

Количество операций фиксации в секунду во время ожидания очистки записей журнала. В идеале это значение должно быть очень низким. Высокие значения указывают на узкое место в пропускной способности журнала.

Имейте в виду, что операции с интенсивным протоколированием могут сильно повлиять на значения счетчиков. Например, при нелимитированном перестроении индекса может очень быстро создаваться огромное количество записей журнала, что перегружает подсистему ввода/вывода и пропускную способность журнала.

Я привык запускать задание агента SQL и собирать данные из представления `sys.dm_io_virtual_file_stats` каждые 5–15 минут, а затем сохранять их в служебной базе данных. Таким образом получается подробная информация о перегрузках ввода/вывода, которая существенно помогает анализировать пропускную способность журнала транзакций и общую производительность операций ввода/вывода.

В любом случае, чтобы улучшить производительность и пропускную способность журнала транзакций, имеет смысл рассмотреть несколько различных факторов. Во-первых, проанализируйте оборудование. Для файлов журнала очень важно использовать быструю дисковую подсистему с малой задержкой. Затем изучите конфигурацию журнала и попытайтесь уменьшить накладные расходы, возникающие из-за большого количества VLF и управления увеличением журнала. Наконец, ищите возможность сократить протоколирование за счет грамотного управления транзакциями и эффективного обслуживания базы данных.

Резюме

SQL Server использует протоколирование с опережающей записью, чтобы поддерживать согласованность базы данных. Записи журнала закрепляются в фай-

ле журнала перед фиксацией транзакций. Недостаточно высокая пропускная способность журнала транзакций и большая задержка ввода/вывода ухудшают производительность системы. По возможности держите журналы транзакций в быстром хранилище с малой задержкой.

Проблемы с пропускной способностью журнала проявляются в ожиданиях `WRITELOG` и `LOGBUFFER`. Когда эти ожидания становятся заметными, стоит устранить неполадки с производительностью журнала. Для этого можно использовать представление `sys.dm_io_virtual_file_stats` и счетчики производительности.

Производительность журнала транзакций можно улучшить, если уменьшить объемы протоколирования, создаваемого базами данных. Для этого продумайте стратегию обслуживания индексов, удалите ненужные индексы, выполните рефакторинг схем баз данных и усовершенствуйте управление транзакциями. Для баз данных, которые обрабатывают много автоматически фиксируемых транзакций и могут выдерживать небольшие потери данных, можно включить отложенную устойчивость.

Убедитесь, что файл журнала сконфигурирован правильно и количество VLF в файле управляемо. Если журнал сконфигурирован неоптимально, попробуйте его перестроить.

В следующей главе я расскажу о группах доступности AlwaysOn и проблемах, с которыми можно столкнуться при их использовании.

Чек-лист устранения неполадок

- ☐ Просмотреть и скорректировать конфигурации журналов транзакций для всех баз данных.
- ☐ Проанализировать количество VLF и при необходимости перестроить журналы.
- ☐ Проверить модели восстановления баз данных и обсудить с заинтересованными сторонами стратегию аварийного восстановления.
- ☐ По возможности уменьшить объемы протоколирования транзакций.
- ☐ Попробовать включить ускоренное восстановление базы данных, если в системе используются длительные активные транзакции и допустимы накладные расходы на ADR.
- ☐ Проанализировать и увеличить пропускную способность журнала транзакций.

ГЛАВА 12

Группы доступности AlwaysOn

Группы доступности AlwaysOn (Always On Availability Groups) — это наиболее распространенная технология *высокой доступности* (*HA, high availability*), используемая в SQL Server. Она поддерживает несколько копий баз данных, из-за чего хранилище перестает быть единой точкой отказа. Также она позволяет масштабировать нагрузку чтения, предоставляя множественные вторичные узлы, доступные для чтения.

В этой главе я приведу обзор внутреннего устройства групп доступности и объясню, как устранять их распространенные проблемы. Вы узнаете о накладных расходах, связанных с синхронными репликами и доступными для чтения вторичными репликами. Наконец, я расскажу о мониторинге групп доступности и о том, как устранять неполадки аварийного переключения.

Обзор групп доступности AlwaysOn

Возможно, самый простой способ объяснить, как работают группы доступности AlwaysOn, — рассмотреть историю этой технологии. Она впервые появилась в SQL Server 2012 в качестве замены и преемника зеркального отображения баз данных, которое раньше называлось кодовым именем *Real Time Log Shipping* (*Доставка записей журнала в реальном времени*). Имя говорит само за себя: и в зеркальном отображении баз данных, и в группах доступности для обмена данными используется поток записей журнала транзакций. Первичный узел отправляет записи журнала вторичным узлам, которые закрепляют их в своих журналах транзакций и воспроизводят изменения в базах данных.

В обеих технологиях предусмотрен один первичный узел, который обрабатывает рабочую нагрузку записи. В SQL Server Enterprise Edition у групп доступности может быть несколько вторичных узлов, из которых некоторые или все могут обрабатывать рабочие нагрузки только для чтения. Возможности Standard Edition ограничены базовыми группами доступности, которые поддерживают только одну вторичную реплику. Эта реплика используется

исключительно для обеспечения высокой доступности и/или аварийного восстановления (DR, disaster recovery). Клиенты не могут подключаться к ней и читать ее данные.

Чтобы обеспечивать высокую доступность, группы доступности поддерживают автоматическое аварийное переключение. Они работают на основе отказоустойчивой кластеризации Windows Server (WSFC, Windows Server Failover Cluster) на Windows и Pacemaker на Linux. До SQL Server 2017 для групп доступности обязательно требовался WSFC. Начиная с SQL Server 2017, их можно использовать без WSFC, однако в этой конфигурации не поддерживается автоматическое аварийное переключение.

На рис. 12.1 показан пример конфигурации группы доступности с тремя узлами в Windows. Прослушиватель группы доступности (Availability Group Listener) — это имя виртуальной сети (аналогично конечной точке кластера WSFC), которое обеспечивает уровень абстракции для подключения клиентов. В этой конфигурации клиенты могут подключаться к группе доступности, не зная, какой узел в данный момент является первичным.

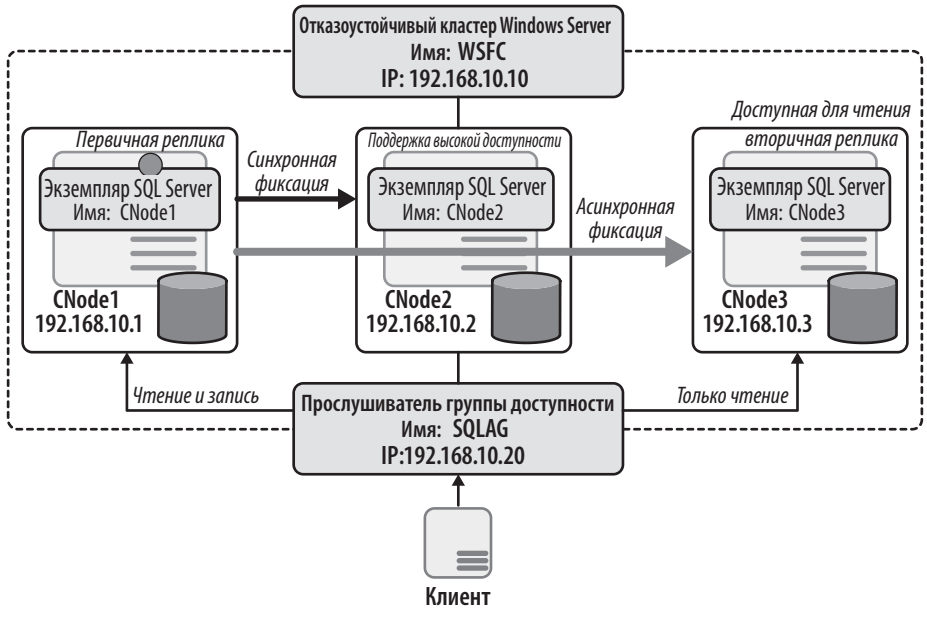


Рис. 12.1. Пример конфигурации группы доступности

Группы доступности работают на уровне группы баз данных. Они реплицируют группу баз данных, переключая их вместе на другой узел в случае аварийного переключения. Каждый узел в топологии хранит собственную копию данных.

Это позволяет избежать ситуации, когда хранилище становится единой точкой отказа.

К сожалению, группы доступности не реплицируют объекты уровня экземпляра. Чтобы система поддерживала высокую доступность и корректно работала после аварийного переключения, на всех узлах нужны соответствующие настройки учетных записей, заданий и других объектов экземпляра. Проверяйте конфигурацию при каждом аудите работоспособности системы и регулярно тестируйте реализацию высокой доступности в промышленных системах.

В библиотеке dbatools[1] с открытым исходным кодом есть много командлетов PowerShell для репликации объектов уровня экземпляра и проверки конфигурации.

Рассмотрим ключевые компоненты инфраструктуры групп доступности: очереди отправки и повтора.

Очереди групп доступности

Узлы группы доступности взаимодействуют друг с другом через поток записей журнала транзакций. Вторичные узлы заносят эти записи в журналы транзакций и асинхронно применяют изменения к базам данных, размещенным на этих узлах.

На рис. 12.2 показан общий вид этого процесса.

Ключевые компоненты — *очередь отправки* и *очередь повтора*.

Очередь отправки (send queue)

Очереди отправки находятся на первичном узле. Они состоят из записей журнала, которые надо отправить на вторичные узлы. На каждом вторичном узле в группе доступности существуют отдельные очереди для каждой базы данных.

Очередь повтора (redo queue)

Очереди повтора находятся на вторичных узлах. Они состоят из записей журнала об изменениях, которые надо применить к базам данных с помощью асинхронного *процесса повтора*. У каждой базы данных на каждом вторичном узле есть собственная очередь повтора.

В исправной группе доступности очереди отправки и повтора остаются настолько короткими, насколько возможно. Длинная очередь отправки увели-

[1] https://dbatools.io/

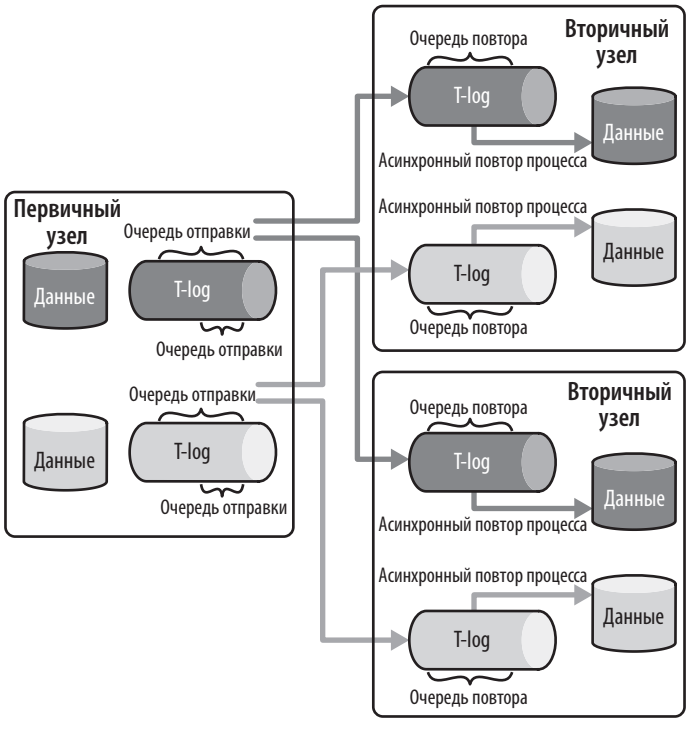


Рис. 12.2. Очереди группы доступности

чивает количество возможных потерь данных в асинхронных репликах и может значительно усугубить блокирование синхронных реплик (подробнее об этом позже в этой главе). Длинная очередь повтора увеличивает время восстановления базы данных и задержку репликации, что, в свою очередь, может повлиять на запросы на доступных для чтения вторичных узлах.

Наконец, длинные очереди отправки и повтора также влияют на усечение журнала транзакций. SQL Server не будет усекать журнал за пределами начальной точки самой длинной очереди отправки. Также (хотя это не задокументировано) он не будет усекать журнал за пределами самой старой начальной точки повтора на всех вторичных узлах.

В листинге 12.1 показан код, с помощью которого можно отслеживать работоспособность групп доступности. Чтобы получить правильные результаты, этот код нужно запускать на первичном узле. Для него требуется SQL Server 2014 или более поздней версии. Версию сценария, работающую на SQL Server 2012, я включил в сопроводительные материалы книги[1].

[1] Azure Data Studio можно скачать с сайта Microsoft: https://oreil.ly/zwwCf.

Листинг 12.1. Код для мониторинга группы доступности

```
SELECT
    ar.replica_server_name as [Replica]
    ,DB_NAME(drs.database_id) AS DB
    ,drs.synchronization_state_desc as [Sync State]
    ,ars.synchronization_health_desc as [Health]
    ,ar.availability_mode as [Synchronous]
    ,drs.log_send_queue_size
    ,drs.redo_queue_size
    ,ISNULL(
        GhostReplicaState.max_low_water_mark_for_ghosts -
            drs.low_water_mark_for_ghosts,0
    ) AS [water_mark_diff]
    ,drs.log_send_rate
    ,drs.redo_rate
    ,pri.last_commit_time AS primary_last_commit_time
    ,IIF(drs.is_primary_replica = 1
        ,pri.last_commit_time
        ,drs.last_commit_time
    ) AS node_last_commit_time
    ,IIF(drs.is_primary_replica = 1
        ,0
        ,DATEDIFF(ms,drs.last_commit_time,pri.last_commit_time)
    ) AS commit_latency
FROM
    sys.availability_groups ag WITH (NOLOCK)
        JOIN sys.availability_replicas ar WITH (NOLOCK) ON
            ag.group_id = ar.group_id
        JOIN sys.dm_hadr_availability_replica_states ars WITH (NOLOCK) ON
            ar.replica_id = ars.replica_id
        JOIN sys.dm_hadr_database_replica_states drs WITH (NOLOCK) ON
            ag.group_id = drs.group_id AND
            drs.replica_id = ars.replica_id
        LEFT JOIN sys.dm_hadr_database_replica_states pri WITH (NOLOCK) ON
            pri.is_primary_replica = 1 AND
            drs.database_id = pri.database_id
        OUTER APPLY
        (
            SELECT MAX(drs2.low_water_mark_for_ghosts) AS
                    max_low_water_mark_for_ghosts
            FROM sys.dm_hadr_database_replica_states drs2 WITH (NOLOCK)
            WHERE drs.database_id = drs2.database_id
        ) GhostReplicaState
WHERE
    ars.is_local = 0
ORDER BY
    replica_server_name, DB;
```

На рис. 12.3 показан вывод листинга 12.1 на одном из моих промышленных серверов.

	Replica	DB	Sync State	Health	Syncronous	log_send_queue_size	redo_queue_size
1			SYNCHRONIZED	HEALTHY	1	0	3605
2			SYNCHRONIZED	HEALTHY	1	0	81
3			SYNCHRONIZED	HEALTHY	1	0	0
4			SYNCHRONIZED	HEALTHY	1	0	0
5			SYNCHRONIZED	HEALTHY	1	0	0
6			SYNCHRONIZED	HEALTHY	1	0	0
7			SYNCHRONIZED	HEALTHY	1	0	0
8			SYNCHRONIZING	HEALTHY	0	60	2898
9			SYNCHRONIZING	HEALTHY	0	0	128

water_mark_diff	log_send_rate	redo_rate	primary_last_commit_time	node_last_commit_time	commit_latency
985	236939	34739	2021-05-13 07:15:08.397	2021-05-13 07:15:05.960	2436
53	0	85281	2021-05-13 07:15:08.340	2021-05-13 07:15:07.680	660
1	0	22799	2021-05-13 07:00:33.760	2021-05-13 07:00:33.760	0
1	0	0	2021-05-13 07:00:34.220	2021-05-13 07:00:34.220	0
1	0	0	2021-05-13 07:00:34.493	2021-05-13 07:00:34.493	0
1	0	0	2021-05-13 07:00:34.777	2021-05-13 07:00:34.777	0
1	0	1	2021-05-13 07:00:35.050	2021-05-13 07:00:35.050	0
38251	0	10584	2021-05-13 07:15:08.397	2021-05-13 07:15:06.483	1913
117	0	50990	2021-05-13 07:15:08.340	2021-05-13 07:15:07.510	830

Рис. 12.3. Вывод сценария мониторинга группы доступности

В первую очередь нужно отслеживать такие параметры:

Работоспособность синхронизации и ее состояние

Сведения о работоспособности синхронизации и ее состоянии приведены в столбцах `synchronization_health_desc` и `synchronize_state_desc`. Они показывают, исправна ли группа доступности и синхронизированы ли данные. Это самые важные показатели, за которыми надо следить.

Длина очередей отправки и повтора

Длину очередей отправки и повтора можно увидеть в столбцах `log_send_queue_size` и `redo_queue_size`. Обе очереди должны быть как можно короче.

Задержка репликации

Чтобы отслеживать задержку репликации, можно сравнивать время последней фиксации на первичном и вторичном узлах. Эти данные доступны в столбцах `primary_last_commit_time` и `node_last_commit_time` соответственно. На задержку влияют обе очереди (отправки и повтора): чем больше данных находится в очередях, тем выше задержка. Очевидно, что задержка должна быть как можно меньше, особенно если вы используете доступные для чтения вторичные реплики.

В представлении `sys.dm_hadr_database_replica_states` есть столбец `secondary_lag_seconds`, но я обнаружил, что его данные менее точны, чем расчет задержки с помощью столбцов `primary_last_commit_time` и `node_last_commit_time`.

Задержка очистки фантомных записей

Длинные очереди отправки и повтора и длительные активные транзакции на доступных для чтения вторичных репликах могут привести к тому, что на основном узле будут откладываться процессы очистки фантомных записей и хранилища версий, что снижает производительность системы. Эту ситуацию можно обнаружить, проанализировав разницу в значениях `low_water_mark_for_ghosts` между первичным и вторичным узлами. В сценарии эта разница отображается в столбце `water_mark_diff`.

Как доступные для чтения вторичные реплики влияют на производительность системы — это важная тема, и мы рассмотрим ее позже в этой главе.

Повторюсь: *крайне важно* следить за работоспособностью и производительностью групп доступности. Многое может пойти не так. В этой главе я расскажу о некоторых характерных проблемах.

Код для мониторинга можно построить на основе листинга 12.1. Метрики, возвращаемые сценарием, можно сравнивать с предопределенными пороговыми значениями, вызывая уведомления по мере необходимости. Настройте пороговые значения в соответствии с вашей конкретной рабочей нагрузкой и инфраструктурой. Например, для асинхронных удаленных реплик пороговые значения для уведомлений очереди отправки могут быть выше, чем для синхронных локальных реплик высокой доступности.

Длину очередей и некоторые другие показатели производительности группы доступности также можно узнать с помощью счетчиков объекта производительности *Database Replica*[1]. На них тоже можно выстраивать систему уведомлений, но этот метод более чувствителен к скачкам нагрузки. Например, большие пакетные операции могут привести к коротким всплескам протоколирования, отчего возникнут ненужные уведомления.

Рассмотрим несколько характерных проблем, с которыми можно столкнуться при работе с группами доступности.

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

Синхронная репликация и опасное ожидание HADR_SYNC_COMMIT

В группах доступности можно настраивать репликацию в синхронном или асинхронном режиме фиксации. Синхронный режим позволяет избежать потери данных за счет дополнительной задержки фиксации.

Если быть точнее, синхронная фиксация позволяет избежать потери данных, только когда группа доступности работоспособна. По умолчанию основной узел продолжает работать даже в случае отказа всех синхронных реплик. Таким образом, если и он откажет, это может привести к потере данных.

В SQL Server 2017 и более поздних версиях с помощью параметра `REQUIRED_SYNCHRONIZED_SECONDARIES_TO_COMMIT` можно указать минимальное количество работоспособных синхронных реплик, необходимых для того, чтобы первичный узел фиксировал транзакции. Если нужного количества реплик нет, то фиксация на основном узле не удастся.

Существует распространенное заблуждение, будто в синхронном режиме данные на вторичных репликах обновляются синхронно с основным узлом. На самом деле синхронно закрепляются только записи журнала, а процесс повтора остается асинхронным и может отставать.

На рис. 12.4 показан поток данных репликации в режиме синхронной фиксации. Как видите, клиент не получает подтверждения, что транзакция зафиксирована, до тех пор пока первичный узел не получит подтверждения, что записи журнала `COMMIT` закреплены на вторичных репликах. Первичный узел ожидает этого подтверждения, генерируя ожидание `HADR_SYNC_COMMIT`.

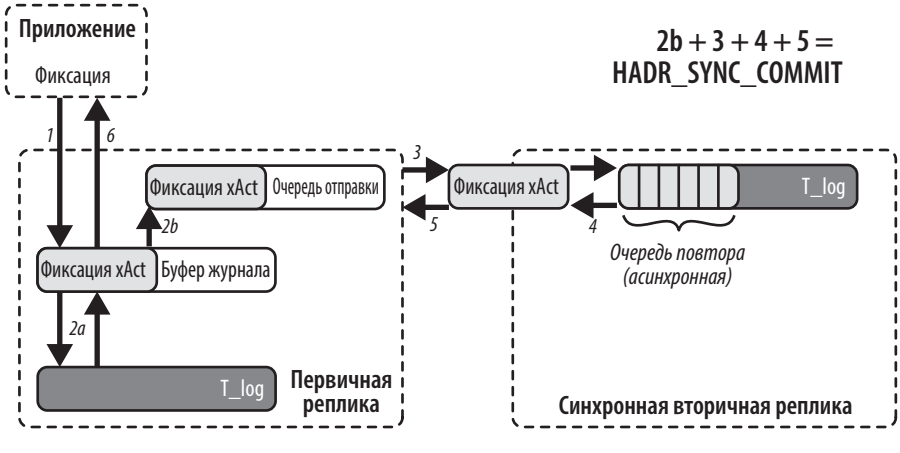


Рис. 12.4. Поток данных при синхронной фиксации

Такое поведение приводит к неочевидным и замысловатым проблемам конкурентного доступа. SQL Server поддерживает транзакции в активном состоянии и не снимает блокировки, пока не получит подтверждения фиксации. Это увеличивает вероятность конкурирующих заявок на блокировку и усугубляет блокирование.

В нормальных условиях увеличение продолжительности транзакций не всегда заметно. Но оно может стать проблемой во время операций с интенсивным протоколированием, когда первичному узлу не хватает пропускной способности, чтобы отправлять записи журнала на вторичные узлы, и очереди отправки начинают расти.

Эту проблему можно воспроизвести[1], если запустить нелимитированное перестроение кластеризованного индекса для крупной таблицы со столбцами *больших объектов* (*LOB*) (таблицы со столбцами больших объектов создают больше записей журнала). На достаточно мощном сервере вы заметите, что очереди отправки для синхронных реплик начнут расти. Это приведет к обширному блокированию в высоконагруженных системах OLTP, большому потреблению доступных рабочих процессов и, вероятно, к отказу системы в скором времени. Еще хуже то, что если отменить перестроение индекса, проблема не решится моментально, потому что первичному узлу все равно придется передавать все записи журнала из очереди отправки.

Ожидание `HADR_SYNC_COMMIT` часто становится одним из основных ожиданий в высоконагруженных системах OLTP. Это может быть штатной ситуацией и не обязательно указывает на проблему. Однако в любом случае нужно оценить, как это ожидание влияет на систему и каковы накладные расходы на синхронную фиксацию.

Изучите среднее время ожидания ресурсов для ожидания `HADR_SYNC_COMMIT` в выходных данных представления `sys.dm_os_wait_stats` (листинг 2.1). Это среднее время, в течение которого SQL Server ожидает подтверждения того, что записи журнала закреплены на вторичных узлах. Оно должно быть как можно меньше, в идеале не более нескольких миллисекунд. Его продолжительность зависит от трех ключевых факторов.

Производительность сети

И записи журнала, и подтверждения проходят через сеть, поэтому хорошая пропускная способность сети и ее низкая задержка *крайне важны* для репликации групп доступности. Чтобы проанализировать и устранить возможные проблемы с сетью, изучите метрики производительности сети `Bytes Received/sec`, `Bytes Sent/sec` и `Current Bandwidth`.

[1] Я подготовил короткое демонстрационное видео на YouTube: https://oreil.ly/nH1MS.

Иногда имеет смысл создать отдельную сеть для групп доступности, отделив репликацию от клиентского трафика. В этом случае надо проверить, что сети физически отделены друг от друга. Если они работают на одних и тех же физических адаптерах локальной сети, эта топология не даст особых преимуществ, потому что трафик из обеих сетей будет направляться в одну и ту же физическую инфраструктуру.

Производительность ввода/вывода на вторичных узлах

Синхронные реплики закрепляют записи в файлах журнала перед тем, как отправить подтверждение первичному узлу. Следовательно, от недостаточной производительности ввода/вывода увеличится задержка фиксации. Посмотрите на простои операций записи журнала в представлении `sys.dm_io_virtual_file_stats`. Для устранения неполадок производительности ввода/вывода можно использовать методы, описанные в главе 3.

Пропускная способность процессора

Как первичному, так и вторичным узлам нужна достаточная пропускная способность ЦП, чтобы обрабатывать репликацию. Убедитесь, что серверы не перегружены, а планировщики равномерно сбалансированы по узлам NUMA (см. листинг 2.4).

Как правило, я не рекомендую выполнять операции чтения на *синхронных* репликах. Клиентские запросы создают дополнительную нагрузку и могут повлиять на скорость репликации, особенно в случае неоптимизированных запросов, которые перегружают подсистему ввода/вывода и таким образом увеличивают задержку записи в журнал. Обычно лучше создавать отдельные асинхронные реплики, с помощью которых можно масштабировать рабочую нагрузку чтения.

Производительность групп доступности можно повысить, если уменьшить количество записей журнала, которые надо обрабатывать. В главе 11 я упомянул несколько способов добиться этого, и здесь они все тоже применимы.

Наконец, если вы все еще работаете на SQL Server 2012 или 2014, рассмотрите возможность обновления до более новой версии. В SQL Server 2016 улучшена производительность многих компонентов, включая группы доступности. Более новые версии SQL Server работают еще лучше.

На рис. 12.5 показаны ожидания на одном промышленном сервере до и после обновления SQL Server до версии 2016. Оба снимка были сделаны в приложении SolarWinds DPA, при этом сервер обрабатывал одну и ту же рабочую нагрузку. Как видите, в SQL Server 2016 ожидания `HADR_SYNC_COMMIT` сократились больше чем на две трети по сравнению с предыдущей версией. Обновление также снизило нагрузку на ЦП на 35 % без каких-либо дополнительных изменений в приложениях.

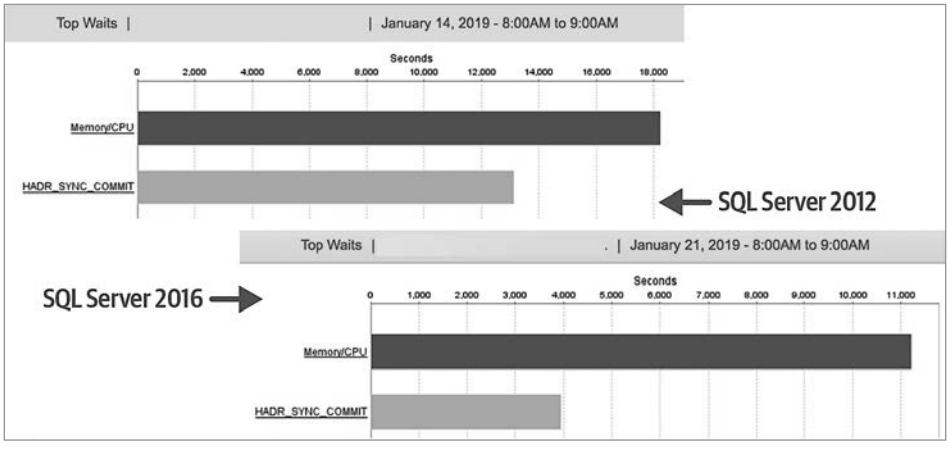


Рис. 12.5. Ожидания до и после обновления SQL Server до версии 2016

Обычно анализ производительности сети, операций ввода/вывода и загрузки ЦП дает достаточно информации, чтобы устранять проблемы с пропускной способностью групп доступности. Но иногда приходится погрузиться глубже и исследовать производительность отдельных операций. Для этого можно использовать расширенные события групп доступности.

Расширенные события групп доступности

SQL Server предоставляет много расширенных событий, которые помогают устранять неполадки с производительностью групп доступности. В табл. 12.1 показаны самые важные из них.

Таблица 12.1. Расширенные события для устранения неполадок с производительностью групп доступности

Расширенное событие	Узел	Описание
`log_flush_start`	Первичный Вторичный	Узел начинает закреплять записи в файлах журнала транзакций
`log_flush_complete`	Первичный Вторичный	Узел заканчивает закреплять записи в файлах журнала транзакций
`hadr_log_block_compression`	Первичный	Первичный узел сжимает блок журнала
`hadr_log_block_decompression`	Вторичный	Вторичный узел распаковывает блок журнала

Расширенное событие	Узел	Описание
hadr_capture_log_block	Первичный	Первичный узел принимает блок журнала для репликации. Поле Mode обозначает состояние: 1: Блок журнала принят 2: Блок журнала поставлен в очередь отправки 3: Блок журнала исключен из очереди и готов к отправке 4: Блок журнала маршрутизирован на соответствующую реплику
ucs_connection_send_msg	Первичный Вторичный	Блок журнала отправлен по сети
hadr_transport _receive_log_block_message	Вторичный	Вторичный узел получает блок журнала. Поле Mode обозначает состояние: 1: Блок журнала получен 2: Блок журнала поставлен в рабочую очередь
hadr_send_harden_lsn_message	Вторичный	Вторичный узел отправляет подтверждение того, что блок журнала закреплен. Поле Mode обозначает состояние: 1: Подтверждение создано 2: Подтверждение готово к отправке 3: Подтверждение маршрутизировано на первичный узел
hadr_lsn_send_complete	Вторичный	Подтверждение отправлено
hadr_receive_harden_lsn _message	Первичный	Первичный узел получает подтверждение. Поле Mode обозначает состояние: 1: Подтверждение получено 2: Подтверждение удалено из очереди и принято в обработку
hadr_db_commit_msg_harden	Первичный	Подтверждение обработано
hadr_db_com mit_mgr_harden_still_waiting	Первичный	Подтверждение фиксации не было получено в течение 2 с. В штатных условиях так быть не должно; это обычно указывает на проблему с группами доступности

На рис. 12.6 показана последовательность событий во время синхронного обмена сообщениями в группе доступности. На первичном узле SQL Server запускает очистку журнала и одновременно отправляет блок журнала вторичному узлу. Вторичный узел декодирует блок, закрепляет его в файле журнала, а затем создает и отправляет подтверждающее сообщение обратно первичному узлу.

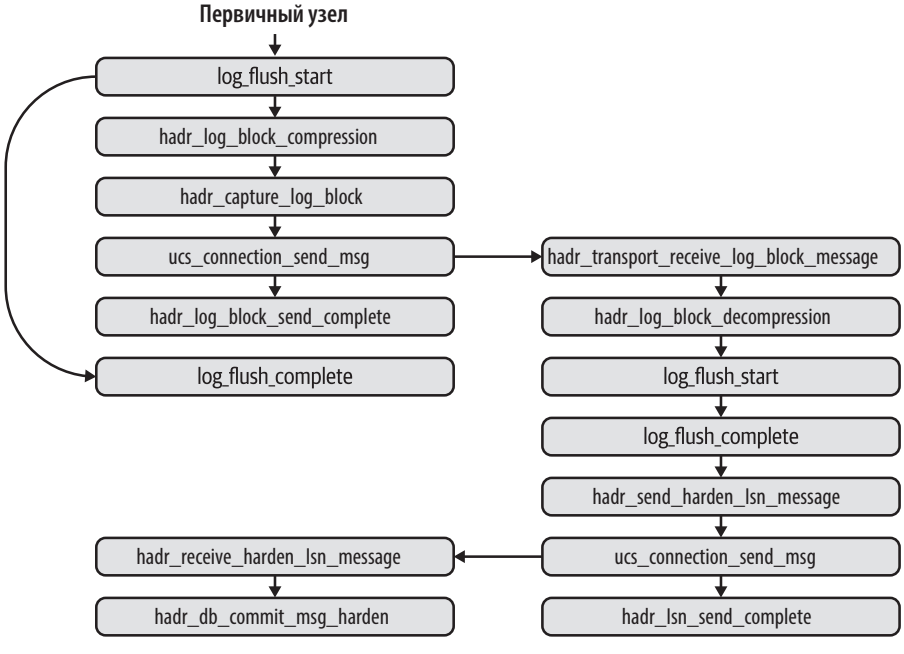


Рис. 12.6. Поток обмена сообщениями в группе доступности в синхронном режиме

В листинге 12.2 показаны два сеанса расширенных событий, которые позволяют регистрировать события, перечисленные в табл. 12.1. Обратите внимание, что эти сеансы могут собирать очень много событий. Не оставляйте их включенными после того, как закончили устранять неполадки.

Здесь не указано событие `ucs_connection_send_msg`, потому что оно создает много шума. Его можно добавить, если вы устраняете возможные проблемы с производительностью сети.

Листинг 12.2. Создание сеансов расширенных событий для устранения неполадок с производительностью группы доступности

```
-- Создать на первичном узле
CREATE EVENT SESSION [AlwaysOn_Tracing_Primary] ON SERVER
ADD EVENT sqlserver.hadr_capture_log_block,
ADD EVENT sqlserver.hadr_db_commit_mgr_harden,
ADD EVENT sqlserver.hadr_db_commit_mgr_harden_still_waiting,
```

```
ADD EVENT sqlserver.hadr_log_block_compression,
ADD EVENT sqlserver.hadr_log_block_send_complete,
ADD EVENT sqlserver.hadr_receive_harden_lsn_message,
ADD EVENT sqlserver.log_flush_complete,
ADD EVENT sqlserver.log_flush_start
ADD TARGET package0.ring_buffer(SET max_events_limit=(0),max_memory=(16384));
GO

-- Создать на вторичном узле
CREATE EVENT SESSION [AlwaysOn_Tracing_Secondary] ON SERVER
ADD EVENT sqlserver.hadr_apply_log_block,
ADD EVENT sqlserver.hadr_log_block_decompression,
ADD EVENT sqlserver.hadr_lsn_send_complete,
ADD EVENT sqlserver.hadr_send_harden_lsn_message,
ADD EVENT sqlserver.hadr_transport_receive_log_block_message,
ADD EVENT sqlserver.log_flush_complete,
ADD EVENT sqlserver.log_flush_start
ADD TARGET package0.ring_buffer(SET max_events_limit=(0),max_memory=(16384));
```

Данные из обоих сеансов можно сопоставлять по полям `log_block_id` и `database_id`. Но тут есть одна загвоздка: в событиях `hadr_send_harden_lsn_message`, `hadr_receive_harden_lsn_message` и `hadr_lsn_send_complete` значение `log_block_id` будет больше, чем в предыдущих событиях. Это связано с тем, как расширенные события собирают данные. Разница в значениях зависит от нагрузки на базу данных, но не должна превышать 120.

На рис. 12.7 показаны события, собранные в моей тестовой среде на основном узле. Можно заметить, что несколько событий перечислены не по порядку. Они все сработали очень быстро и поэтому имеют одинаковую временну́ю метку.

На рис. 12.8 показаны события, собранные на вторичном узле. Временные метки немного расходятся с первичным узлом, потому что мне не удалось идеально синхронизировать время между серверами.

name	timestamp	mode	log_block_id
hadr_capture_log_block	2021-05-10 19:09:31.4952867	1	158913901495
hadr_capture_log_block	2021-05-10 19:09:31.4952867	2	158913901495
log_flush_start	2021-05-10 19:09:31.4952867	*NULL*	158913901495
hadr_log_block_compression	2021-05-10 19:09:31.4952867	*NULL*	158913901495
hadr_log_block_send_complete	2021-05-10 19:09:31.4952867	*NULL*	158913901495
log_flush_complete	2021-05-10 19:09:31.4952867	*NULL*	158913901495
hadr_receive_harden_lsn_message	2021-05-10 19:09:31.4962860	1	158913901496
hadr_receive_harden_lsn_message	2021-05-10 19:09:31.4962860	2	158913901496
hadr_db_commit_msg_harden	2021-05-10 19:09:31.4962860	*NULL*	*NULL*

Рис. 12.7. Расширенные события на первичном узле

name	timestamp	mode	log_block_id
hadr_transport_receive_log_block_message	2021-05-10 19:09:31.5116162	1	158913901495
hadr_transport_receive_log_block_message	2021-05-10 19:09:31.5116664	2	158913901495
hadr_log_block_decompression	2021-05-10 19:09:31.5116888	NULL	158913901495
hadr_log_block_decompression	2021-05-10 19:09:31.5116967	NULL	158913901495
log_flush_start	2021-05-10 19:09:31.5117582	NULL	158913901495
log_flush_complete	2021-05-10 19:09:31.5123777	NULL	158913901495
hadr_send_harden_lsn_message	2021-05-10 19:09:31.5124051	1	158913901496
hadr_send_harden_lsn_message	2021-05-10 19:09:31.5124295	2	158913901496
hadr_send_harden_lsn_message	2021-05-10 19:09:31.5124316	3	158913901496
hadr_lsn_send_complete	2021-05-10 19:09:31.5125995	NULL	158913901496

Рис. 12.8. Расширенные события на вторичном узле

Временные метки отдельных операций помогают обнаружить возможные узкие места. Например, медленная дисковая подсистема на вторичном узле приведет к задержке между событиями `log_flush_start` и `log_flush_complete`. Из-за недостаточной мощности процессора может увеличиться время событий `hadr_log_block_compression` и `hadr_log_block_decompression`, если используется сжатие. Проанализируйте данные и сверьте их с другими метриками.

Сжатие в разных версиях SQL Server ведет себя по-разному. SQL Server 2012 и 2014 по умолчанию сжимают весь трафик группы доступности. Однако в SQL Server 2016 и более поздних версиях сжатие используется только при асинхронном обмене данными, а не при синхронном. Но расширенные события сжатия и распаковки все равно регистрируются, даже если сами действия не выполняются.

Существуют три флага трассировки, с помощью которых можно изменить режим сжатия в SQL Server 2016 и более поздних версиях.

`T9592`

Этот флаг включает сжатие трафика между синхронными репликами. Его можно включить, если приходится работать с синхронными репликами в медленной сети. Имейте в виду, что сжатие повышает нагрузку на ЦП на обоих узлах и может увеличить задержку `HADR_SYNC_COMMIT` в быстрых сетях.

`T1462`

Этот флаг отключает сжатие трафика между асинхронными репликами. Это может снизить загрузку ЦП узлов в очень высоконагруженных системах OLTP за счет дополнительного сетевого трафика.

`T9567`

При автоматическом заполнении новых узлов в группах доступности SQL Server не использует сжатие. Флаг `T9567` позволяет его включить. Это может ускорить процесс автоматического заполнения, однако создаст дополнительную нагрузку на ЦП основного узла.

Обратите внимание, что эти три флага трассировки могут сместить рабочую нагрузку и узкие места от ЦП к сети и наоборот, поэтому используйте их с осторожностью!

Наконец, в современных версиях SSMS есть инструменты, которые предоставляют аналогичные данные о задержках. Они вызываются командой Collect Latency Data (Собирать данные о задержках) на контрольной панели группы доступности. Эта команда создает и запускает сеансы расширенных событий на узлах, собирающих события группы доступности. Эти сеансы работают в течение двух минут и сохраняют данные в целевой файл. После этого SQL Server обрабатывает данные и предоставляет доступ к ним через стандартные отчеты во всплывающем меню группы доступности в обозревателе объектов SSMS.

SSMS собирает данные с некоторыми ограничениями. Во-первых, этот метод работает, только если вы подключаетесь к серверам с использованием аутентификации Windows и имеете права системного администратора на всех репликах. Во-вторых, при этом нужно, чтобы на всех серверах работал агент SQL Server. В-третьих, здесь используются целевые файлы, что может увеличить нагрузку на высоконагруженные серверы с интенсивным трафиком групп доступности. Но все равно собирать данные через SSMS часто оказывается проще, чем вручную анализировать расширенные события.

Асинхронная репликация и доступные для чтения вторичные реплики

В отличие от синхронного режима фиксации, в асинхронном режиме первичный узел не ожидает подтверждения того, что записи журнала закреплены на вторичных репликах. Транзакция фиксируется, когда запись журнала `COMMIT` сохраняется в журнале транзакций первичного узла.

На рис. 12.9 показан поток данных репликации в асинхронном режиме. От длины очереди отправки на основном узле зависит возможный объем потери данных в случае отказа SQL Server.

В SQL Server Enterprise Edition можно масштабировать рабочую нагрузку чтения, выполняя запросы на вторичных узлах. Это отличная функция, которая помогает повысить пропускную способность системы и снижает нагрузку на первичный узел. Но есть несколько факторов, которые стоит учитывать.

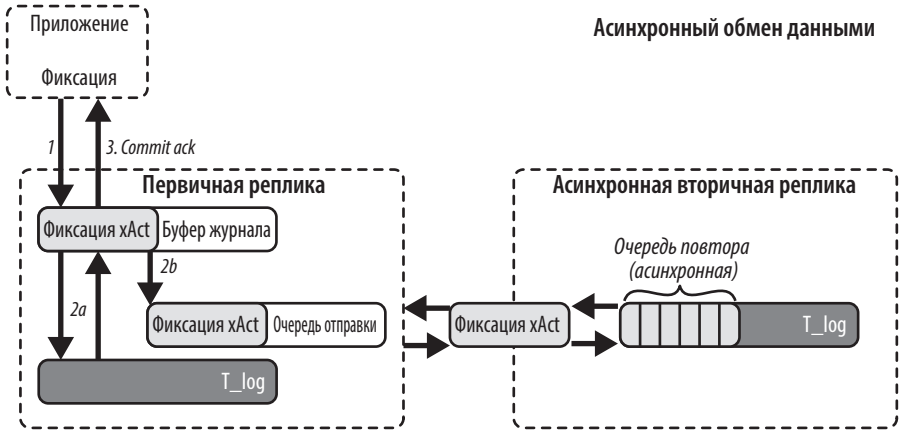


Рис. 12.9. Поток данных при асинхронной фиксации

Как я уже упоминал, следует соблюдать осторожность, направляя запросы на узлы, которые используют синхронную репликацию. Накладные расходы на запросы могут увеличить задержку репликации и величину ожиданий `HADR_SYNC_COMMIT`, что повлияет на первичный узел. Всегда лучше оставить синхронные узлы в покое и создавать асинхронные реплики, с помощью которых масштабировать рабочую нагрузку чтения.

Независимо от режима репликации данные на вторичных репликах всегда будут отставать от первичного узла, потому что процесс повтора, применяющий изменения к базе данных, является асинхронным. В нормальных условиях отставание бывает очень небольшим — порядка миллисекунд или даже микросекунд, однако оно может увеличиваться во время операций с интенсивным протоколированием, таких как обслуживание индекса или обработка больших пакетов.

Не используйте вторичные узлы для критических запросов, которым нужны актуальные данные. Нельзя гарантировать, что отставание всегда будет небольшим. Также не следует разделять по разным узлам запросы чтения и записи в рамках одной бизнес-транзакции, потому что результаты окажутся несогласованными.

Давайте теперь рассмотрим другую, менее известную проблему, связанную с доступными для чтения вторичными репликами. Эта проблема может затронуть первичный узел совершенно неожиданным образом.

Особенности доступных для чтения вторичных реплик

Для запросов на вторичных узлах SQL Server всегда использует уровень изоляции `SNAPSHOT`, игнорируя инструкцию `SET TRANSACTION ISOLATION LEVEL` и соответствующие указания таблиц. На уровне `SNAPSHOT` операции чтения не бло-

кируются операциями записи. Это происходит, даже если не включать параметр базы данных ALLOW_SNAPSHOT_ISOLATION.

Уровень изоляции SNAPSHOT также означает, что SQL Server задействует управление версиями строк на вторичных узлах. Как вы помните из главы 8, при этом SQL Server использует хранилище версий в tempdb, а также создает 14-байтовые указатели хранилища версий в измененных строках данных.

В инфраструктуре групп доступности на первичном и вторичном узлах базы данных абсолютно одинаковы, поэтому невозможно поддерживать управление версиями строк на одних лишь вторичных узлах. Чтобы базы данных были идентичны, SQL Server приходится выделять место для 14-байтовых указателей хранилища версий на первичном узле, даже если в базе данных отключены оптимистичные уровни изоляции.

SQL Server не использует хранилище версий tempdb на первичных узлах, если не включен параметр READ_COMMITTED_SNAPSHOT или ALLOW_SNAPSHOT_ISOLATION. Тем не менее во время модификации данных он добавляет по 14 байт к строкам данных, что может привести к дополнительным разбиениям страниц и фрагментации индекса.

К сожалению, с доступными для чтения вторичными репликами связан еще один, менее известный эффект: длительные транзакции SNAPSHOT на вторичных узлах могут откладывать выполнение задач по очистке фантомных записей и хранилища версий на первичном узле. Такие транзакции работают со снимком данных на момент начала транзакции. SQL Server не может очистить удаленные строки и повторно использовать пространство, потому что транзакции SNAPSHOT может потребоваться доступ к старым версиям строк.

То же самое относится к длинным очередям отправки и повтора. SQL Server не может очистить удаленные строки, потому что вторичные реплики могут запустить транзакцию SNAPSHOT до того, как воспроизводить очистку фантомных записей журнала. Это может стать проблемой, если реплика отключается или постоянно запаздывает, применяя изменения.

Посмотрим на код в листинге 12.3. Он создает две таблицы в базе данных. Таблица T1 содержит 65 536 строк и использует 65 536 страниц: по одной строке на страницу данных.

Листинг 12.3. Доступные для чтения вторичные реплики: создание таблицы

```
CREATE TABLE dbo.T1
(
    ID INT NOT NULL,
    Placeholder CHAR(8000) NULL,
    CONSTRAINT PK_T1 PRIMARY KEY CLUSTERED(ID)
);

CREATE TABLE dbo.T2
```

```
(
    Col INT
);
;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 строки
,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 строки
,N3(C) AS (SELECT 0 FROM N2 AS T1 CROSS JOIN N2 AS T2) -- 16 строк
,N4(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N3 AS T2) -- 256 строк
,N5(C) AS (SELECT 0 FROM N4 AS T1 CROSS JOIN N4 AS T2 ) -- 65,536 строк
,IDS(ID) AS (SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) FROM N5)
INSERT INTO dbo.T1(ID)
    SELECT ID FROM IDS;
```

Запустим транзакцию на вторичном узле и выполним запрос к таблице `T2`, как показано в листинге 12.4. Я использую явные транзакции, но поведение будет аналогичным, если в автоматически фиксируемой транзакции есть долго выполняющаяся инструкция.

Листинг 12.4. Доступные для чтения вторичные узлы: запуск транзакции на вторичном узле

```
BEGIN TRAN
    SELECT * FROM dbo.T2;
```

Затем, используя код из листинга 12.5, удалим все данные из таблицы `T1` и запустим запрос, который выполняет просмотр кластеризованного индекса на первичном узле.

Листинг 12.5. Доступные для чтения вторичные узлы: удаление данных и просмотр кластеризованного индекса

```
DELETE FROM dbo.T1;

-- Подождать 1 минуту
WAITFOR DELAY '00:01:00.000';

SET STATISTICS IO ON
SELECT COUNT(*) FROM dbo.T1;
SET STATISTICS IO OFF
--Вывод: Table 'T1'. Scan count 1, logical reads 65781
```

Хоть таблица и пуста, страницы данных не освободились. Это приводит к накладным расходам ввода/вывода и множеству логических операций чтения на первичном узле.

Наконец, изучим статистику индекса, используя код из листинга 12.6.

Листинг 12.6. Доступные для чтения вторичные узлы: проверка статистики индекса

```
SELECT
    index_id, index_level, page_count
    ,record_count, version_ghost_record_count
FROM
```

```
sys.dm_db_index_physical_stats
(
    DB_ID()
    ,OBJECT_ID(N'dbo.T1')
    ,1
    ,NULL
    ,'DETAILED'
);
```

На рис. 12.10 показан результат запроса. На листовом уровне индекса оказалось 65 536 строк в столбце `version_ghost_record_count`. Это фантомные строки, которые невозможно удалить из-за активных транзакций, использующих управление версиями строк. В нашем случае такая транзакция выполняется на другом (вторичном) узле.

	index_id	index_level	page_count	record_count	version_ghost_record_count
1	1	0	65536	0	65536
2	1	1	243	65536	0
3	1	2	1	243	0

Рис. 12.10. Статистика индекса

Такое поведение может увеличить нагрузку на ввод/вывод и ЦП, поскольку во время выполнения запроса SQL Server приходится просматривать фантомные строки данных. При этом также увеличивается размер `tempdb`, потому что хранилище версий не очищается.

Влияние на ввод/вывод и загрузку ЦП может быть особенно высоким, если система реализует обработку сообщений на основе очередей в базе данных. Таблицы, в которых хранятся сообщения, обычно невелики, но данные в них крайне изменчивы. Фантомные записи быстро накапливаются, увеличивая нагрузку на ЦП и ввод/вывод.

Вот в каком сценарии я впервые столкнулся с этой проблемой. Одна из таблиц использовалась для обработки сообщений и проводила около 100 вставок и удалений каждую секунду. Неуправляемая транзакция отчетов на вторичном узле увеличила загрузку ЦП первичного узла на 30 % за ночь без каких-либо изменений в рабочей нагрузке системы. К тому моменту, когда я обнаружил проблему, в таблице было больше миллиона страниц данных, хранящих всего несколько строк. Запросы, которые читали сообщения, просматривали все эти страницы данных, увеличивая загрузку ЦП.

К сожалению, многие поручают неоптимизированные отчетные запросы вторичным узлам, не понимая возможных последствий. Помните об этом и отслеживайте задержку очистки фантомных записей с помощью столбца `water_mark_diff` из

листинга 12.1. Установите пороговое значение для уведомлений в зависимости от рабочей нагрузки вашей системы и с учетом накладных расходов на отложенную очистку фантомных записей.

И последнее, но не менее важное: не включайте доступные для чтения вторичные реплики, если вы не обращаетесь к ним с запросами. Они влияют на производительность и увеличивают стоимость лицензирования, поэтому не стоит активировать эту функцию, если вы ее не используете.

Параллельный повтор

В SQL Server 2012 и 2014 процесс повтора в группах доступности задействовал один поток для каждой базы данных. Из-за этого в высокопроизводительных системах OLTP с чрезвычайно интенсивной модификацией не хватало пропускной способности повторного выполнения, что затрудняло использование групп доступности как таковых.

Начиная с SQL Server 2016, процесс повтора может быть параллельным, если для воспроизведения записей журнала SQL Server использует несколько потоков. Количество потоков, выполняющих процесс повтора, зависит от количества процессоров на сервере и от количества баз данных в группе доступности.

Если в группе доступности несколько баз данных, только первые шесть используют параллельный повтор. Их порядок обычно зависит от того, в какой момент времени базы данных вошли в группу доступности. К сожалению, это поведение плохо документировано и может измениться в будущих версиях SQL Server.

В SQL Server 2016 и 2017 параллельный повтор не использовался с базами данных In-Memory OLTP. Это ограничение было снято в SQL Server 2019.

Параллельный повтор — отличная функция, *когда она работает*, а это происходит не всегда. Я видел немало случаев, когда параллельный повтор внезапно переставал работать, отчего производительность процедуры повтора значительно снижалась. Обычно это происходит в высоконагруженных системах OLTP, где есть доступные для чтения вторичные реплики.

Типичный симптом этой проблемы — повышенное потребление ЦП и постоянно растущие очереди повтора. При этом часто наблюдаются ожидающие задачи, как правило, с одним или несколькими из этих типов ожидания в выходных данных представлений `sys.dm_os_waiting_tasks` и `sys.dm_exec_requests`:

- `DIRTY_PAGE_TABLE_LOCK`
- `DPT_ENTRY_LOCK`

- PARALLEL_REDO_TRAN_TURN
- PARALLEL_REDO_FLOW_CONTROL

Отключить параллельный повтор можно с помощью флага трассировки уровня сервера T3459, запустив команду DBCC TRACEON(3459,-1). Это можно сделать «на лету», не перезапуская сервер. Однако, чтобы отключить этот флаг и вернуться к параллельному повтору, понадобится перезапуск в сборках до SQL Server 2017 CU9, SQL Server 2016 SP2 CU2 и SQL Server 2016 SP1 CU10.

Как обычно, рекомендую устанавливать последние накопительные обновления SQL Server. В них содержится множество исправлений, связанных с параллельным повтором, особенно в SQL Server 2016 и 2017.

Устранение неполадок аварийного переключения

Автоматическое аварийное переключение — отличная функция, которая улучшает высокую доступность и добавляет еще несколько задач. Когда происходит незапланированное аварийное переключение, нужно найти его основную причину. В других случаях, наоборот, имеет смысл проанализировать, почему аварийное переключение *не* сработало так, как нужно.

Стратегия устранения неполадок одинакова в обоих случаях: собрать информацию и проанализировать ее после аварийного переключения. Источники этой информации я опишу позже в этом разделе. Сперва давайте в целом рассмотрим, как SQL Server взаимодействует с отказоустойчивой кластеризацией Windows Server.

Группы доступности и отказоустойчивая кластеризация Windows Server

Автоматическое аварийное переключение в группах доступности реализовано через службы отказоустойчивой кластеризации Windows Server (WSFC). Группы доступности становятся кластерным ресурсом в кластере, который управляет ими аналогично другим службам. Это значит, что если у WSFC возникнут проблемы — например, если он потеряет кворум, — это также повлияет на соответствующие группы доступности.

В управлении ресурсами кластера есть две ключевые проверки: IsAlive и IsHealthy. С их помощью кластер часто проверяет, что ресурс доступен в сети и работает должным образом.

Если проверка IsAlive не удалась, кластер может выполнить аварийное переключение или завершить работу экземпляра SQL Server. Он также может инициировать эти действия либо по результатам проверки IsHealthy, либо если

обнаружится много ошибок нарушения доступа, либо если сервер потребляет ресурсы чрезвычайно интенсивно и продолжительно.

Обе эти проверки выполняет библиотека ресурсов SQL Server, которая постоянно взаимодействует с экземпляром SQL Server. Проверка `IsAlive` осуществляется по протоколу Shared Memory, который позволяет двум процессам совместно использовать память для обмена данными. За частоту передачи данных отвечает свойство `LeaseTimeout` в ресурсе кластера группы доступности. По умолчанию для `LeaseTimeout` установлено значение 20 000 мс. Проверка `IsAlive` запускается каждые 5 секунд, что составляет всего 25 % от этого значения.

Механизм аренды (lease) существует только на основном узле и представляет собой, по сути, «пульс» группы доступности. Когда кластер не получает подтверждения, что аренда активна, он считает, что ее срок истек и группа доступности неработоспособна. Он перестает принимать запросы на запись, чтобы избежать так называемого *«раздвоения личности»*, то есть состояния, когда несколько узлов ведут себя как первичная реплика, принимая запросы на запись. Затем группа доступности пытается выполнить аварийное переключение, предполагая, что сам WSFC исправен.

В то же время проверка `IsHealthy` опирается на хранимую процедуру `sp_server_diagnostics`, которая предоставляет информацию о работоспособности системы в целом и группы доступности, а также о состоянии компонентов SQL Server и потреблении ресурсов. За частоту этой проверки отвечает свойство `HealthCheckTimeout` в ресурсе кластера группы доступности. По умолчанию его значение равно 30 000 мс. Частота проверки `IsHealthy` составляет одну треть этого значения, то есть 10 с.

Другое свойство, `FailureConditionLevel`, определяет, при каком условии проверка работоспособности считается неудачной. Оно принимает следующие значения:

1: `OnServerDown`

   Проверка работоспособности подтверждает, что группа доступности доступна в сети. Это удается, если проверка `IsAlive` проходит успешно.

2: `OnServerUnresponsive`

   Проверка завершается неудачно, если в срок, указанный в `HealthCheckTimeout`, не получены данные от процедуры `sp_server_diagnostics`.

3: `OnCriticalServerError`

   Проверка завершается неудачно, если какой-либо из основных компонентов SQL Server возвращает ошибку. Это значение по умолчанию.

4: `OnModerateServerError`

   Проверка завершается неудачно, если компонент ресурса возвращает ошибку.

5: `OnAnyQualifiedFailureConditions`

Проверка завершается неудачно, если компонент обработки запросов возвращает ошибку.

Нетрудно догадаться, что чем больше значение, тем чаще будет запускаться аварийное переключение. В большинстве случаев не стоит изменять `FailureConditionLevel`, хотя его можно временно уменьшить, когда надо устранить неполадки аварийного переключения, возникшие из-за чрезвычайно интенсивного использования ресурсов. Я не рекомендую увеличивать этот параметр, если вы не хотите, чтобы аварийное переключение происходило при первых признаках возможной проблемы, даже если она может решиться сама собой.

`LeaseTimeout` можно повысить, если возникли проблемы со связью между узлами кластера. Максимально допустимое значение этого свойства можно определить по формуле:

`LeaseTimeout` <= (`SameSubnetDelay` * `SameSubnetThreshold`) * 2.

`SameSubnetDelay` и `SameSubnetThreshold` — это свойства уровня кластера, которые указывают частоту опроса кластера («частоту пульса») и количество неудачных опросов, после которого кластер будет считаться неработоспособным. Эти параметры, как и `LeaseTimeout`, можно регулировать.

Параметры `CrossSubnetDelay` и `CrossSubnetThreshold` управляют «пульсом» в межподсетевом кластере. Убедитесь, что значения `SameSubnetDelay` и `SameSubnetThreshold` не превышают их.

Увеличьте `HealthTimeout`, если вы не хотите аварийного переключения в ситуации, когда из-за коротких всплесков потребления ресурсов SQL Server перестает отвечать на запросы. Например, программное обеспечение для виртуализации и резервного копирования виртуальных машин может приостанавливать виртуальные машины во время резервного копирования. В этом случае можно увеличить `HealthTimeout`, чтобы избежать аварийных переключений. Однако будьте осторожны: это может задержать аварийное переключение в кластере в других допустимых ситуациях.

Как я уже отмечал, в SQL Server для Linux аварийное переключение опирается на внешнюю службу Pacemaker (это может измениться в будущих версиях SQL Server). Концептуально Pacemaker ведет себя аналогично WSFC, хотя SQL Server не может обмениваться с ним данными напрямую. Реализация основана на механизме опроса, при котором Pacemaker регулярно опрашивает состояние SQL Server и баз данных.

Устранение неполадок аварийного переключения

Почему вообще происходят аварийные переключения? Возможно, вы уже догадались о некоторых причинах. Вот наиболее распространенные из них:

- Проблемы с оборудованием.
- Проблемы с WSFC, такие как потеря кворума кластера из-за неполадок с сетью или подключением.
- Истечение срока аренды, которое может быть вызвано неполадками WSFC, такими как чрезвычайно высокая нагрузка на сервер или зависание ОС.
- Истекшее время ожидания проверки работоспособности. Это часто бывает из-за высокой нагрузки SQL Server (ЦП и/или диска), большого количества ошибок нарушения доступа и дампов потоков, зависания ОС или виртуальной машины или из-за других факторов.
- Человеческие ошибки при настройке или обслуживании WSFC.

К счастью, SQL Server и ОС предоставляют много информации, которая помогает выяснить причину проблемы. Например:

Сеанс расширенного события `AlwaysOn_health`

Сеанс расширенного события `AlwaysOn_health` запускается при подготовке групп доступности. Этот сеанс содержит набор событий, которые отслеживают работоспособность групп доступности, изменения состояния и ролей узлов, а также операции DDL, связанные с группами доступности. В него также входит несколько событий для мониторинга критических ошибок: например, событие `availability_group_lease_expired` происходит по истечении срока аренды.

Сеанс `AlwaysOn_health` хорошо подходит для того, чтобы с него начинать устранение неполадок. Он может не показать основную причину инцидента, но даст информацию о том, что и когда произошло с группой доступности, а также какие действия были предприняты. Наконец, он позволяет точно выявить ситуации, когда ручное аварийное переключение произошло либо преднамеренно, либо в результате человеческой ошибки.

Файлы `SQLDIAG`

Как уже отмечалось, в WSFC используется хранимая процедура `sp_server_diagnostics` как часть проверки кластера `IsHealthy`. Эта процедура предоставляет сведения о работоспособности компонентов SQL Server и использовании ресурсов, таких как память и ЦП, а также о состоянии монитора взаимных блокировок, нарушениях доступа и дампах потоков. Она также дает общую информацию о работоспособности всех групп доступности на сервере.

Данные из `sp_server_diagnostics` перехватываются скрытым сеансом расширенного события и сохраняются в файлах XEL в папке журнала SQL Server. Эти файлы чрезвычайно полезны при устранении неполадок аварийного переключения, потому что из них можно узнать, были ли некоторые компоненты сервера перенагружены или неисправны. Имя каждого файла состоит из имен сервера и экземпляра, за которыми следует строка `SQLDIAG`.

Сеанс расширенного события `system_health`

Сеанс расширенного события `system_health` — это еще один сеанс, который создается и запускается в SQL Server по умолчанию. Он предоставляет информацию об общем состоянии SQL Server и критических ошибках, а также сведения о состоянии компонентов и диагностические данные. Эти диагностические данные аналогичны информации в файлах `SQLDIAG`, однако метрики агрегируются с бóльшими интервалами в пять минут.

Журнал кластера

Журнал кластера — важнейший источник данных для устранения неполадок. Он содержит подробную информацию об ошибках, которые привели к аварийному переключению, и может помочь выявить кворум и другие проблемы, связанные с кластером.

Журнал SQL Server

Журнал SQL Server содержит информацию о критических ошибках, состоянии аварийных переключений и реплик группы доступности. При устранении неполадок полезно посмотреть в журнале, какие события происходили во время инцидента.

Системные журналы

Системные журналы и журнал событий Windows могут содержать сведения о критических состояниях и ошибках системы, таких как аппаратные сбои.

Обратите внимание, что SQL Server и WSFC ротируют журналы и файлы расширенных событий. Имеет смысл собирать данные как можно скорее после аварийного переключения, иначе они могут исчезнуть. Собрав данные, проанализируйте их целостно, чтобы определить проблему.

Когда аварийное переключение не происходит

В некоторых случаях нужно устранить противоположную проблему: почему группа доступности не выполнила аварийного переключения, хотя должна была это сделать. Вот несколько распространенных причин этой неполадки.

Неправильная конфигурация

Группа доступности настроена неправильно: например, в ней не заданы параметры автоматического аварийного переключения или есть базы данных, вошедшие в группу доступности только на одном узле.

Проблемы со связью

Аварийного переключения может не произойти, если первичный узел потеряет связь с вторичным, на который он должен переключиться. Для переключения также требуется, чтобы базы данных были полностью синхронизированы, поэтому оно может не произойти, если вторичный узел не «догнал» репликацию, когда стал вновь доступен после отключения.

Проблемы с кластером

Автоматическому аварийному переключению могут помешать некоторые проблемы WSFC: например, если библиотека ресурсов SQL Server потеряет соединение с SQL Server.

Группа доступности превысила максимальное количество аварийных переключений

У каждого ресурса кластера есть набор свойств, которые определяют максимальное количество аварийных переключений, допустимых в течение определенного периода. В группах доступности это значение по умолчанию равно $N - 1$, где N — количество узлов в кластере. Период по умолчанию составляет шесть часов.

Эти пороговые значения можно регулировать в настройках кластерного ресурса группы доступности.

Подход к устранению неполадок в этом случае очень похож на тот, который применяется для выявления нежелательных аварийных переключений: просмотрите данные из журналов кластера и SQL Server, файлов `SQLDIAG` и сеансов `AlwaysOn_health` и `system_health` и воссоздайте обстоятельства, которые имели место во время инцидента. Обычно в перечисленных источниках достаточно данных, чтобы помочь понять причину проблемы.

Резюме

Группы доступности AlwaysOn — самая популярная технология высокой доступности, используемая в SQL Server. Она позволяет избежать ситуации, когда хранилище становится единой точкой отказа, причем в Enterprise Edition можно масштабировать рабочую нагрузку чтения между несколькими репликами.

Группы доступности не реплицируют объекты уровня экземпляра, такие как учетные записи и задания. Синхронизируйте их на узлах, чтобы системы могли работать после аварийных переключений.

Репликация в группах доступности основана на потоке записей журнала транзакций. Отслеживайте очереди отправки и повтора и настройте уведомления на случай, когда они чрезмерно вырастут. Длинные очереди могут привести к потере данных и увеличить время восстановления. Они также могут помешать усечению журнала и вызвать другие проблемы с производительностью.

Синхронный режим фиксации помогает избежать потери данных, но при этом возникает дополнительная задержка фиксации во время репликации. Проанализируйте время ожидания `HADR_SYNC_COMMIT` и сократите его, насколько возможно.

Доступные для чтения вторичные реплики позволяют масштабировать рабочую нагрузку чтения. Однако они могут задержать очистку фантомных записей и журнала версий на первичном узле, увеличивая на нем нагрузку на ЦП и ввод/вывод. Избегайте длительных транзакций на вторичных репликах и следите за их рабочей нагрузкой.

При устранении неполадок аварийного переключения просмотрите данные из сеансов расширенных событий `AlwaysOn_health` и `system_health`, файлы `SQLDIAG`, а также журналы ОС, SQL Server и кластера. Обычно они содержат достаточно информации, чтобы диагностировать проблему.

В следующей главе я рассмотрю еще несколько распространенных типов ожидания.

Чек-лист устранения неполадок

- ☐ Проверить, что объекты уровня экземпляра синхронизированы между узлами группы доступности.
- ☐ По возможности протестировать аварийное переключение и высокую доступность.
- ☐ Настроить мониторинг группы доступности и уведомления о длине очередей, задержке репликации, отставании очистки фантомных записей и событиях аварийного переключения.
- ☐ Проанализировать влияние задержки фиксации, если используется синхронная фиксация. Изучить время ожидания ресурса `HADR_SYNC_COMMIT` и при необходимости устранить неполадки с производительностью репликации.
- ☐ Проверить, включены ли доступные для чтения вторичные реплики. Оценить, насколько запросы только для чтения влияют на доступные для чтения синхронные реплики.
- ☐ Отключить доступные для чтения вторичные реплики, если они не используются.

ГЛАВА 13

Другие примечательные типы ожиданий

В этой довольно короткой главе описаны несколько типов ожиданий, которые пока не упоминались. Начнем с типа ASYNC_NETWORK_IO, который возникает, когда клиент недостаточно быстро принимает данные от SQL Server. Затем я расскажу об ожидании THREADPOOL и опасном состоянии нехватки рабочих потоков. После этого рассмотрим типы ожидания, связанные с резервным копированием, и способы повышения его производительности.

Глава заканчивается обзором OLEDB и нескольких других типов вытесняющего ожидания, которые возникают, когда SQL Server вызывает API ОС для переключения в режим выполнения с вытеснением.

Ожидания ASYNC_NETWORK_IO

ASYNC_NETWORK_IO — это распространенный тип ожидания, который встречается почти в каждой системе. Неопытные специалисты смотрят на имя ожидания и думают, будто оно связано с плохой производительностью сети. На самом деле его причина шире: ожидание возникает, когда SQL Server ждет, пока клиентское приложение примет данные.

Конечно, медленная сеть может вызвать это состояние, но чаще всего дело в неэффективной конструкции клиентского приложения. Если оно считывает и обрабатывает данные построчно, то SQL Server вынужден ждать.

В листинге 13.1 показан образец кода, который вызовет эту проблему. Клиентское приложение принимает и обрабатывает строки по одной, оставляя SqlDataReader открытым. Рабочий процесс SQL Server ожидает, пока клиент примет все строки, и генерирует ожидание ASYNC_NETWORK_IO.

Листинг 13.1. Код, генерирующий ожидание ASYNC_NETWORK_IO

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    SqlCommand command = new SqlCommand(cmdText, connection);
    connection.Open();
    using (SqlDataReader reader = command.ExecuteReader())
    {
        while (reader.Read())
        ProcessRow((IDataRecord)reader);
    }
}
```

Правильным подходом здесь было бы сначала как можно скорее прочитать все строки, а затем обработать их, как показано в листинге 13.2. Если размер данных очень велик и у клиента недостаточно памяти, чтобы их кэшировать, то на стороне клиента может потребоваться пакетная обработка.

Листинг 13.2. Код, уменьшающий ожидание ASYNC_NETWORK_IO

```
List<Orders> orderRows = new List<Orders>();
using (SqlConnection connection = new SqlConnection(connectionString))
{
    SqlCommand command = new SqlCommand(cmdText, connection);
    connection.Open();
    using (SqlDataReader reader = command.ExecuteReader())
    {
        while (reader.Read())
        orderRows.Add(ReadOrderRow((IDataRecord)reader));
    }
}
ProcessAllOrderRows(orderRows);
```

Мой любимый способ доказать, что построчная обработка приводит к ожиданию ASYNC_NETWORK_IO, — запустить небольшую демонстрацию в SSMS. Подключитесь к локальному экземпляру SQL Server: при этом SSMS будет использовать протокол *Shared Memory*, не связанный с сетевым трафиком. Затем очистите статистику ожидания и запустите инструкцию SELECT *, которая выбирает из таблицы несколько тысяч строк. Когда она выполнится, на верхней строчке списка ожиданий вы увидите ASYNC_NETWORK_IO, несмотря на полное отсутствие сетевого трафика.

Теперь включите параметр «Discard results after execution» («Отбрасывать результаты после выполнения») на панели Tools ▸ Options ▸ Query Results ▸ SQL Server ▸ Results to Grid и повторите тест. Вы убедитесь, что ожидания ASYNC_NETWORK_IO больше нет. В первом тесте оно появилось из-за неэффективности SSMS, которая заполняет сетку результатов построчно. Эта реализация работает медленно, так что SQL Server ждет, пока каждая строка отобразится в сетке, и генерирует ожидание.

Если это ожидание наблюдается в больших объемах, прежде всего проанализируйте производительность сети. Проверьте топологию сети (помните, что пропускная способность сети зависит от самого медленного компонента), а также показатели производительности сети.

Если вы убедились, что дело не в производительности сети, проанализируйте клиентские приложения. Неэффективный код можно найти с помощью представлений `sys.dm_os_waiting_tasks`, `sys.dm_exec_requests`, `sys.dm_exec_sessions` и `sys.dm_exec_connections`. Правда, изменить клиентский код не всегда возможно.

Также обратите внимание на производительность клиентских приложений. Бывает, что ожидание `ASYNC_NETWORK_IO` генерируют приложения, работающие на перегруженных серверах или неправильно настроенных виртуальных машинах, у которых не хватает пропускной способности, чтобы обрабатывать нагрузку.

В моей практике ожидания `ASYNC_NETWORK_IO` не всегда создают проблемы. Я их игнорирую, если они не слишком велики, а система работает под умеренной нагрузкой.

Но не стоит думать, будто это ожидание совершенно безвредно. Оно отнимает ресурсы исполнителей на сервере и увеличивает нагрузку на память и ЦП. Из-за него SQL Server может дольше удерживать блокировки строк, что приводит к проблемам блокирования. Впрочем, обычно это не самая большая из всех неполадок, и целесообразнее сфокусироваться на борьбе с другими очагами снижения производительности.

Ожидания THREADPOOL

В отличие от ожиданий `ASYNC_NETWORK_IO`, на ожидания `THREADPOOL` нужно обращать внимание, даже если их мало. Они указывают на то, что у SQL Server недостаточно рабочих потоков, чтобы назначать их задачам. Когда это происходит, клиенты не могут подключиться к SQL Server и получают ошибки вида *connection timeout* («Время ожидания сервера истекло»), как если бы SQL Server вообще был отключен.

Одна из ситуаций, когда это может случиться, — наличие длинных цепочек блокирования, обычно с блокировками модификации схемы (`Sch-M`). Рассмотрим пример, который имитирует это состояние.

Для начала уменьшим количество исполнителей на сервере, запустив код из листинга 13.3. Этот код также активирует удаленное выделенное административное соединение (DAC); подробнее об этом позже.

Не выполняйте этот сценарий на промышленных серверах, потому что он нарушит их работоспособность. После завершения теста верните значение max worker threads в 0.

Листинг 13.3. Включение удаленного выделенного административного соединения и сокращение числа исполнителей

```
EXEC sp_configure N'max worker threads', 128;
GO
EXEC sp_configure 'remote admin connections', 1;
GO
RECONFIGURE
GO
```

Затем создадим маленькую таблицу и вставим в нее строку, которая устанавливает интентную монопольную блокировку (`IX`). Запустите код из листинга 13.4, но не фиксируйте транзакцию и не закрывайте сеанс после этого.

Листинг 13.4. Ожидание THREADPOOL: создание тестовой таблицы и установка интентной монопольной блокировки

```
CREATE TABLE dbo.ThreadPoolDemo
(
    Col1 INT
);
GO

BEGIN TRAN
    INSERT INTO dbo.ThreadPoolDemo VALUES(0);
```

Теперь давайте откроем другой сеанс и запустим инструкцию `ALTER TABLE`, как показано в листинге 13.5. Эта инструкция будет заблокирована, потому что блокировки (`Sch-M`) и (`IX`) несовместимы.

Листинг 13.5. Ожидание THREADPOOL: изменение таблицы

```
-- Запускайте НЕ в том же сеансе, что листинг 13.4
ALTER TABLE dbo.ThreadPoolDemo ADD Col2 INT;
```

Теперь любые другие сеансы, пытающиеся получить доступ к таблице, будут заблокированы из-за того, что в очереди есть заявка на блокировку (`Sch-M`). Они окажутся приостановлены и будут ждать, занимая ресурсы исполнителей на сервере.

Чтобы воспроизвести эту ситуацию, запустим пакетный сценарий Windows, показанный в листинге 13.6. Он открывает несколько сеансов, которые пытаются получить данные из таблицы. Измените имя сервера и базы данных в соответствии с вашей средой.

Листинг 13.6. Ожидание THREADPOOL: создание нескольких сеансов и блокирование

```
@ECHO OFF
SET query="SELECT * FROM SQLServerInternals.dbo.ThreadPoolDemo"
SET p1=-S
SET server=.
SET p3=-E
SET p4=-Q

FOR /L %%I IN (1,1,150) DO (
    START "" "sqlcmd.exe" %p1% %server% %p3% %p4% %query%
)
```

Этот код займет всех доступных исполнителей. Если теперь вы попытаетесь подключиться к серверу из SSMS, то получите ошибку подключения, показанную на рис. 13.1: ведь в системе больше нет доступных исполнителей, которые обработали бы подключение.

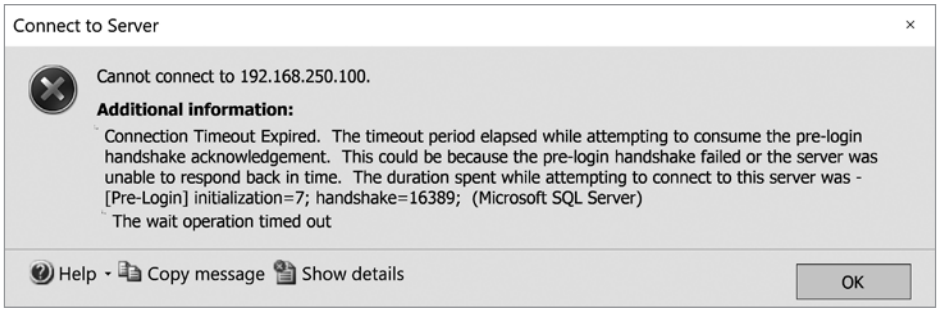


Рис. 13.1. Ошибка подключения из-за исчерпания пула рабочих процессов

Когда возникает такая ситуация (и в других случаях, когда сервер перестает отвечать на запросы), к серверу можно подключиться через выделенное административное соединение (DAC, dedicated admin connection). SQL Server резервирует частный планировщик и небольшой объем памяти для DAC, чтобы оставить возможность для устранения неполадок.

Чтобы подключиться к SQL Server через DAC, используйте префикс `ADMIN`: в имени сервера в поле подключения к SSMS или параметр `-A` в утилите sqlcmd. Подключаться могут только обладатели роли сервера `sysadmin`, причем в каждый момент времени только один сеанс может использовать DAC. Не подключайтесь к DAC в обозревателе объектов SSMS, для которого требуется собственное подключение. Кроме того, отключите функцию IntelliSense в окне запроса перед подключением. В старых версиях SSMS вы можете получить безобидное сообщение об ошибке подключения IntelliSense, даже если она отключена.

По умолчанию DAC доступен только локально. Иногда сервер бывает перегружен настолько, что ОС перестает отвечать, так что нельзя использовать DAC. Во

время первоначальной настройки сервера всегда включайте удаленный доступ к DAC, как я сделал в листинге 13.3.

Попробуем подключиться к серверу через DAC и запросить представление `sys.dm_os_waiting_tasks` с помощью кода, показанного в листинге 13.7. Обратите внимание, что я отфильтровал из выходных данных системные сеансы, чтобы сосредоточиться на цепочке блокирования и ожиданиях `THREADPOOL`. Если бы это была реальная работа по устранению неполадок, я бы не стал этого делать.

Листинг 13.7. Ожидание `THREADPOOL`: запрос представления `sys.dm_os_waiting_tasks`

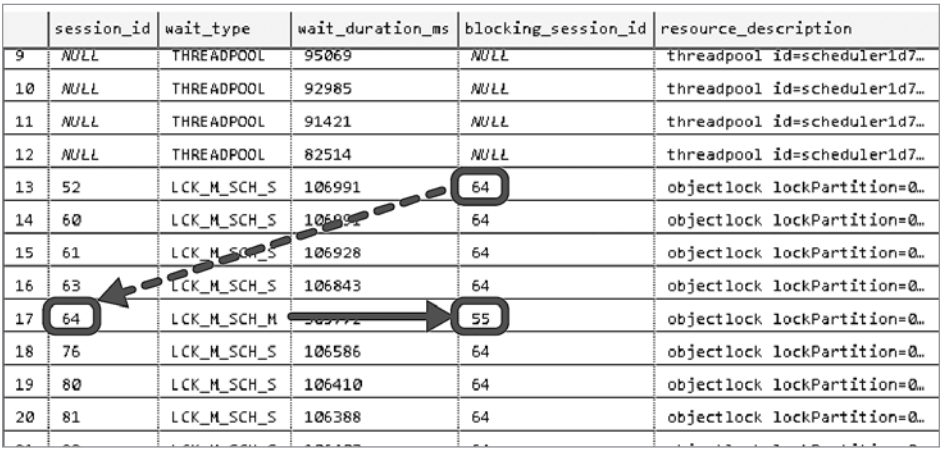
```
SELECT
    session_id
    ,wait_type
    ,wait_duration_ms
    ,blocking_session_id
    ,resource_description
FROM
    sys.dm_os_waiting_tasks WITH (NOLOCK)
WHERE
    (session_id > 50 OR session_id IS NULL) AND
    wait_type NOT IN (N'CLR_AUTO_EVENT',N'QDS_ASYNC_QUEUE'
        ,N'XTP_PREEMPTIVE_TASK',N'BROKER_RECEIVE_WAITFOR'
        ,N'QDS_PERSIST_TASK_MAIN_LOOP_SLEEP')
ORDER BY
    session_id,
    wait_duration_ms DESC;
```

На рис. 13.2 показан вывод кода. Вы видите много сеансов, которые ожидают блокировки стабильности схемы для таблицы, генерируя тип ожидания `LCK_M_SCH_S`. Они блокируются сеансом, содержащим инструкцию `ALTER TABLE`, из-за несовместимости блокировок (`Sch-S`) и (`Sch-M`). В свою очередь, `ALTER TABLE` блокируется сеансом со `SPID=55`, который выполнил инструкцию `INSERT` в незафиксированной транзакции. Чтобы решить проблему, можно принудительно завершить с помощью команды `KILL` либо корневой блокирующий сеанс, либо сеанс `ALTER TABLE`.

Как вы, возможно, заметили, в выходных данных нет корневого блокирующего сеанса со `SPID=55`. В этом сеансе выполнялась неуправляемая незафиксированная транзакция, и в нашем примере он не был заблокирован. В представлении `sys.dm_exec_sessions` этот сеанс можно увидеть в спящем режиме с параметром `open_transaction_count=1`. С помощью листинга 2.3 можно получить информацию о сеансе и клиентском приложении, чтобы продолжать устранение неполадок.

Некоторые задачи с ожиданием `THREADPOOL` также отображаются в представлении `sys.dm_os_waiting_tasks`. Эти задачи относятся к запросам на подключение от клиентов. У SQL Server нет доступных исполнителей для обработки

таких запросов, что в конечном счете вызывает ошибки подключения на стороне клиента.



	session_id	wait_type	wait_duration_ms	blocking_session_id	resource_description
9	NULL	THREADPOOL	95069	NULL	threadpool id=scheduler1d7…
10	NULL	THREADPOOL	92985	NULL	threadpool id=scheduler1d7…
11	NULL	THREADPOOL	91421	NULL	threadpool id=scheduler1d7…
12	NULL	THREADPOOL	82514	NULL	threadpool id=scheduler1d7…
13	52	LCK_M_SCH_S	106991	64	objectlock lockPartition=0…
14	60	LCK_M_SCH_S	106891	64	objectlock lockPartition=0…
15	61	LCK_M_SCH_S	106928	64	objectlock lockPartition=0…
16	63	LCK_M_SCH_S	106843	64	objectlock lockPartition=0…
17	64	LCK_M_SCH_M	365772	55	objectlock lockPartition=0…
18	76	LCK_M_SCH_S	106586	64	objectlock lockPartition=0…
19	80	LCK_M_SCH_S	106410	64	objectlock lockPartition=0…
20	81	LCK_M_SCH_S	106388	64	objectlock lockPartition=0…

Рис. 13.2. Вывод представления `sys.dm_os_waiting_tasks`

Также стоит отметить, что у этих задач пустые значения `session_id`, потому что сеансы еще не инициализированы. Поэтому они не будут отображаться в выводе представления `sys.dm_exec_requests`.

Бывают и другие ситуации, которые приводят к ожиданиям `THREADPOOL`. Например, такие ожидания могут возникать из-за нехватки памяти или из-за ненадлежащего оснащения серверов. Количество рабочих процессов в системе зависит от объема памяти и других факторов, и может оказаться так, что для текущей нагрузки не хватает исполнителей. Посмотрите, нет ли в журнале ошибок признаков нехватки памяти и дампов SQL Server.

Также возможно, что сервер не справляется с нагрузкой из-за того, что на нем слишком много активных сеансов или одновременных запросов с параллельными планами. В этом случае стоит снизить эту активность и, возможно, отрегулировать систему, чтобы уменьшить общую нагрузку.

Проверьте настройку максимального количества исполнителей (max worker thread). Иногда это значение по ошибке оказывается слишком низким. Установите значение по умолчанию (0) и посмотрите, как это повлияет на систему.

Возможно, вам захочется увеличить максимальное количество исполнителей, но это редко решает проблему. Например, массовое блокирование может быстро появиться снова, и новые исполнители тоже окажутся заблокированы. Не паникуйте! В этом случае, как всегда, лучше устранить первопричину проблемы.

Ожидания, связанные с резервным копированием

Как можно догадаться по именам, ожидания BACKUPIO и BACKUPBUFFER происходят во время резервного копирования и восстановления. Они указывают на плохую пропускную способность операций, когда SQL Server не удается достаточно быстро записывать или читать файлы резервных копий. В начале процесса настройки производительности я редко занимаюсь именно этими ожиданиями, но если их чересчур много, стоит проанализировать несколько факторов, описанных далее.

Ожидания, связанные с резервным копированием, вводом/выводом и сетью, довольно часто встречаются вместе, если относятся к одним и тем же ресурсам. Например, медленная или перегруженная дисковая подсистема может порождать BACKUPIO, PAGEIOLATCH, WRITELOG и другие ожидания, связанные с вводом/выводом. В таких случаях ожидания, связанные с резервным копированием, могут служить дополнительным подтверждением проблемы.

Тем не менее, если ожидания проявляются на одних и тех же ресурсах, проверьте, влияет ли операция резервного копирования на другие компоненты. Например, увеличивается ли во время этой операции количество и среднее время ожиданий PAGEIOLATCH, WRITELOG или HADR_SYNC_COMMIT?

Многие инструменты мониторинга собирают информацию об ожиданиях, происходящих в определенные промежутки времени, и ее можно использовать для анализа. Кроме того, можно сохранять данные sys.dm_os_wait_stats в служебной базе данных через регулярные промежутки времени — возможно, с помощью задания агента SQL Server. В сопутствующих материалах этой книги приведен сценарий, который позволяет получить моментальный снимок статистики ожидания за некоторый заданный интервал времени. Этот сценарий можно использовать в вашей реализации мониторинга.

Наконец, если ожидания, связанные с резервным копированием, ощутимо влияют на производительность SQL Server, попробуйте реорганизовать процесс резервного копирования. Используйте разностные резервные копии, задавайте разные местоположения для хранения файлов и корректируйте расписание. Ваша стратегия и доступные варианты действий зависят от инфраструктуры, требований к целевому времени восстановления (RTO, recovery time objective) и целевой точке восстановления (RPO, recovery point objective), от версии и выпуска SQL Server и других факторов.

Повышение производительности резервного копирования

Механизмы резервного копирования, встроенные в SQL Server, предлагают не очень широкий выбор параметров конфигурации. Но кое-что можно под-

регулировать, чтобы настроить и улучшить производительность резервного копирования. Давайте посмотрим, что именно.

Сжатие резервных копий

SQL Server позволяет сжимать файлы резервных копий. В большинстве случаев это уменьшает размер этих файлов за счет дополнительной нагрузки на ЦП, вызванной сжатием.

Обычно это хороший компромисс. Чем меньше файлы резервных копий, тем меньше времени требуется для их передачи по сети, что сократит время восстановления в случае аварии. Также чем меньше эти файлы, тем меньше нагрузка на дисковую подсистему и необходимый объем хранилища в ней.

Как правило, если на сервере достаточно процессорных мощностей, сжатие резервных копий лучше включить. Есть еще несколько соображений, связанных с прозрачным шифрованием данных (TDE, transparent data encryption), которые я вскоре рассмотрю.

Чередование резервных копий

Можно разделить резервную копию базы данных на несколько файлов и создать *резервную копию с чередованием (striped backup)*. Это позволит распараллелить операции резервного копирования и восстановления, и SQL Server будет выполнять их в нескольких потоках.

Эта функция очень полезна для больших баз данных и может значительно сократить время резервного копирования и восстановления. Но имейте в виду, что она ресурсоемка и добавляет нагрузку на инфраструктуру. Не устанавливайте количество отрезков чередования больше количества логических ядер ЦП на сервере и контролируйте накладные расходы, которые может вызвать резервное копирование.

Нужно также проанализировать узкие места процесса резервного копирования. Например, если вы создаете резервную копию базы данных в сетевом расположении, а в сети высокие задержки или у нее недостаточная пропускная способность, то чередование резервных копий и распараллеливание процесса мало чем помогут. В этом случае лучше попробовать чередовать резервные копии в локальном хранилище с прямым подключением (DAS) и копировать файлы в сетевое расположение позже, когда резервное копирование будет завершено.

Параметры BUFFERCOUNT и MAXTRANSFERSIZE

SQL Server позволяет регулировать количество буферов ввода/вывода (опция `BUFFERCOUNT`) и максимальный размер блока передачи (опция `MAXTRANSFERSIZE`)

в процессе резервного копирования. Обычно увеличение обоих параметров ускоряет процесс, но настраивать их следует аккуратно. Когда скорость растет, операция резервного копирования начинает потреблять больше памяти, что может повлиять на другие компоненты SQL Server. Если неправильно задать параметры, это чревато даже тем, что вся доступная память закончится.

Если вы хотите сжать базу данных с прозрачным шифрованием данных (TDE) в SQL Server 2016 или 2017, установите параметр `MAXTRANSFERSIZE` так, чтобы он был больше 65 536 байт (64 Кбайт). В результате SQL Server переключится на новый улучшенный алгоритм сжатия, который работает с шифрованием. Без этого сжатие зашифрованной базы данных не сэкономит много места. В SQL Server 2019 и более поздних версиях эта настройка не требуется, потому что там `MAXTRANSFERSIZE` автоматически регулируется при сжатии баз данных с поддержкой TDE.

Частичные резервные копии базы данных

Когда вы работаете с большими базами данных, часто значительная часть данных со временем перестает меняться: представьте себе работу с таблицами, предназначенными только для добавления, или ситуацию, когда данные через некоторое время становятся доступны только для чтения.

В этом случае можно секционировать данные, используя секционированные таблицы и/или представления. Статическую часть данных можно поместить в отдельные файловые группы, пометив их как «только для чтения». Для таких файловых групп можно один раз создать резервные копии, а затем исключить эти группы из регулярного полного (`FULL`) резервного копирования, что сэкономит время и значительно уменьшит размер резервной копии на диске. (Подробнее об этой реализации можно узнать в моем блоге или в моей книге «Pro SQL Server Internals» (Apress, 2016).)

В конечном итоге выбранный метод настройки процессов резервного копирования и восстановления должен соответствовать стратегии аварийного восстановления вашей организации. Изучите требования и политики RTO и RPO и разработайте реализацию, которая будет их поддерживать.

HTBUILD и другие ожидания с префиксом HT*

Ожидания `HTBUILD`, `HTDELETE`, `HTMEMO`, `HTREINIT` и `HTREPARTITION` возникают во время управления внутренними хеш-таблицами в пакетном режиме выполнения. До SQL Server 2019 выполнение в пакетном режиме применялось почти исключительно с индексами columnstore. В SQL Server 2019 и более поздних версиях его также можно использовать с таблицами на основе строк.

В небольших количествах эти ожидания не обязательно свидетельствуют о проблеме. Однако они могут быть признаком плохого обслуживания индексов columnstore. В частности, они иногда указывают на наличие больших несжатых разностных хранилищ или множества небольших групп строк разного размера. Просмотрите индексы columnstore и перестройте все неэффективные секции. Для анализа можно использовать представление `sys.column_store_row_group`[1].

Я еще не видел проблем с этими ожиданиями в пакетном режиме выполнения с таблицами на основе строк в SQL Server 2019. Однако предполагаю, что эти ожидания могут возникнуть из-за неточной статистики.

Наконец, я хотел бы отметить, что документация Microsoft предлагает уменьшить `MAXDOP` или увеличить порог стоимости для параллелизма, чтобы снизить эти ожидания. На мой взгляд, это неправильный подход: он замаскирует проблему и отключит выполнение в пакетном режиме для некоторых запросов, тем самым снизив производительность.

Вытесняющие ожидания

Как вы помните из главы 2, в ОС SQL Server используется вытесняющее планирование. Исполнители добровольно уступают управление, когда истекает выделенный им квант процессорного времени, позволяя работать другим исполнителям.

Но бывают исключения, когда SQL Server должен вызвать внешнюю функцию, которую он не контролирует. Представьте себе вызов API ОС для аутентификации пользователя на контроллере домена или вызов расширенной хранимой процедуры. Когда это происходит, SQL Server переключает рабочий процесс в режим вытесняющего выполнения и больше не контролирует его планирование. Рабочий процесс остается в состоянии `RUNNING`, но при этом также генерирует ожидание одного из вытесняющих типов, которых в SQL Server 2019 более 200. Большинство из них не связаны с состязаниями, и их можно игнорировать. Но о некоторых ожиданиях стоит знать, и сейчас мы их рассмотрим.

Тип ожидания PREEMPTIVE_OS_WRITEFILEGATHER

Ожидания `PREEMPTIVE_OS_WRITEFILEGATHER` происходят во время процесса нулевой инициализации. Как вы помните, SQL Server всегда инициализирует файлы журналов нулями, а также может инициализировать нулями файлы данных, если мгновенная инициализация файлов не включена.

[1] https://oreil.ly/BK5CL

Увидев это ожидание, проверьте и включите мгновенную инициализацию файлов, предоставив учетной записи запуска SQL Server разрешение «Выполнение задач управления томами» («Perform volume management tasks») (`SE_MANAGE_VOLUME_NAME`). Помните, что при этом возникает небольшая угроза безопасности (см. главу 1), хотя для большинства систем это не страшно.

Также посмотрите на параметры автоувеличения журнала транзакций. Если файлы журнала увеличиваются большими фрагментами, это может растянуть время инициализации нулями и породить неэффективную структуру виртуального файла журнала (VLF). Как вы узнали в главе 11, размером журнала транзакций лучше управлять вручную.

Очевидно, стоит убедиться, что у вас нет запланированных процессов, которые регулярно сокращали бы журналы транзакций. Как вы уже знаете, это плохая практика.

Тип ожидания PREEMPTIVE_OS_WRITEFILE

Ожидания `PREEMPTIVE_OS_WRITEFILE` могут указывать на узкое место во время синхронной записи в файлы. Этот тип ожидания редко вызывает проблемы, но может быть признаком медленной или перегруженной дисковой подсистемы на сервере.

Когда я вижу этот тип ожидания в больших объемах, я проверяю, не выполняет ли сервер множественные трассировки SQL или аудиты SQL, используя файлы в качестве целей для сохранения данных. Также это ожидание может быть связано с записью снимков базы данных, созданных пользователями, или внутренних снимков, созданных операцией `DBCC CHECKDB`.

Типы ожидания, связанные с аутентификацией

Есть несколько типов ожидания, возникающих во время вызовов аутентификации пользователя, когда SQL Server ждет ответов от контроллера домена. Это типы ожиданий с именами, начинающимися на `PREEMPTIVE_OS_AUTH*`, а также ожидания `PREEMPTIVE_OS_LOOKUPACCOUNTSID` и `PREEMPTIVE_OS_ACCEPTSCURITYCONTEXT`.

Они могут быть связаны с инфраструктурой. Типичный пример — аутентификация облачных серверов SQL Server на удаленных локальных контроллерах Active Directory. Задержка вызовов может продлить процесс аутентификации, что приводит к высокому проценту связанных с ней ожиданий.

Другая возможная причина — код, который выполняется не в том контексте выполнения, что вызвавший его сеанс. Бывает, что модулю `EXECUTE AS OWNER` или `EXECUTE AS USER` требуются постоянные вызовы аутентификации, которые могут дорого обойтись при большой нагрузке.

Если вы видите ожидания, связанные с аутентификацией, проверьте работоспособность инфраструктуры Active Directory и задержку вызовов аутентификации, а затем изучите, как часто эти вызовы выполнялись.

Ожидания OLEDB

`OLEDB` — это еще один тип вытесняющего ожидания, который возникает, когда SQL Server ожидает данных от поставщика OLE DB. Чаще всего это ожидание наблюдается в следующих случаях:

- вызовы на связанные серверы;
- выполнение некоторых пакетов SQL Server Integration Services (SSIS);
- операции во время выполнения `DBCC CHECKDB`;
- запросы к динамическим административным представлениям.

У ожиданий из первых двух категорий обычно относительно высокое время ожидания. Они указывают на длительные удаленные запросы и долгое выполнение пакетов SSIS. Попробуйте устранить неполадки с производительностью этих вызовов на удаленных серверах и/или пересмотрите логику пакета SSIS.

Ожидания, порожденные операцией `DBCC CHECKDB` и доступом к DMV, обычно короткие и в среднем длятся не более нескольких миллисекунд. Суммарные показатели дадут представление о накладных расходах на эти операции. Чтобы устранить накладные расходы на `DBCC CHECKDB`, можно перенести эти процедуры на резервный сервер проверки.

Ожидание доступа к DMV указывает на дополнительную нагрузку от инструментов мониторинга, которые постоянно запрашивают эти представления. Если при этом возникают слишком длительные ожидания, попробуйте изменить стратегию мониторинга и/или переключиться на другие инструменты.

Наконец, еще одна известная проблема с ожиданием `OLEDB` возникает, если удаленный связанный сервер (не SQL Server) не завершает соединение должным образом. SQL Server оставляет соединение открытым, создавая бесконечное ожидание `OLEDB` для сеанса. К сожалению, это нельзя исправить никаким простым способом, кроме перезапуска SQL Server.

Типы ожидания: подводим итоги

В SQL Server существуют сотни типов ожидания, и охватить их все невозможно. Да и смысла в этом нет, потому что с большинством из них вы наверняка никогда не столкнетесь.

В этой книге мы рассмотрели наиболее распространенные типы ожидания, которых достаточно, чтобы устранять большинство неполадок, но на практике вы можете встретить и другие типы. Пускай это вас не пугает!

Сначала изучите тип ожидания. Начать можно с документации Microsoft[1]. Еще один полезный ресурс — SQLSkills Wait Types Library[2]. Ну и Google с Bing в помощь, конечно же; иногда и там можно найти ценную информацию.

Условия, при которых появляется тот или иной тип ожидания, укажут вам на проблемные области. Сейчас вы уже знаете достаточно много о модели выполнения и компонентах SQL Server, чтобы выстраивать стратегию устранения неполадок и решать проблемы. Просто не забывайте смотреть на проблему целостно и избегать узкого видения. Все компоненты SQL Server работают во взаимосвязи друг с другом, и возникающие неполадки могут касаться многих из них.

Резюме

Ожидания `ASYNC_NETWORK_IO` происходят, когда SQL Server ожидает, пока клиентское приложение примет данные. Этот тип ожидания может указывать на плохую пропускную способность сети или на проблемы с клиентскими приложениями, которые считывают и обрабатывают данные построчно.

Ожидания `THREADPOOL` — признак опасного состояния нехватки рабочих потоков. Когда это происходит, у SQL Server не хватает исполнителей для выполнения клиентских запросов, и он перестает отвечать. Причиной такого состояния могут быть длинные цепочки блокирования, недостаточный объем памяти (из-за неправильной конфигурации или чрезвычайной нехватки памяти) и ресурсоемкая параллельная нагрузка (часто неоптимизированная).

Когда SQL Server не отвечает и перестает принимать обычные соединения, можно использовать выделенное административное соединение, чтобы устранить текущие неполадки. Обязательно включите удаленный доступ к DAC во время подготовки сервера.

Ожидания `BACKUPIO` и `BACKUPBUFFER` возникают, если SQL Server не хватает пропускной способности для записи или чтения из файлов резервных копий. Увидев эти типы ожидания, проанализируйте инфраструктуру и отрегулируйте процесс резервного копирования.

Вытесняющие ожидания возникают, когда SQL Server вызывает внешние функции, переходя на модель вытесняющего выполнения. Большинство этих ожиданий можно игнорировать, но стоит обратить внимание на ожидания, свя-

[1] https://oreil.ly/xEUg4

[2] https://oreil.ly/ZHKse

занные с аутентификацией, ожидания OLEDB и вытесняющие типы ожидания, относящиеся к вводу/выводу.

В следующей главе мы сменим тему и обсудим, как диагностировать неэффективность схемы базы данных и индексации.

Чек-лист устранения неполадок

- ☐ Если наблюдается много ожиданий ASYNC_NETWORK_IO, проверить топологию сети и реализацию клиента.
- ☐ Исследовать ожидания THREADPOOL, если они есть.
- ☐ Проанализировать стратегии аварийного восстановления и резервного копирования и отрегулировать процесс резервного копирования, если требуется.
- ☐ Если наблюдается HTBUILD или другие ожидания, связанные с выполнением в пакетном режиме, проанализировать размер разностных хранилищ и состояние групп строк в индексах columnstore.
- ☐ Устранить неполадки с ожиданиями вытесняющих типов и OLEDB, если они слишком заметны.

ГЛАВА 14

Анализ схемы базы данных и индексов

До сих пор большинство задач по устранению неполадок, описанных в этой книге, рассматривали базы данных и пользовательские приложения как «черные ящики». Мы занимались такими методами повышения производительности, которые не требуют изменений в базах данных и приложениях, кроме индексации и простых правок кода T-SQL. Такой подход проще и быстрее оправдывает затраченные усилия, но ограничивает диапазон результатов, которых можно достичь.

Не поймите меня неправильно: зачастую и впрямь можно добиться *достаточно хороших* результатов, не внося существенных изменений в базы данных и приложения. Но будет нелишним в общих чертах проанализировать схему базы данных и использование индекса и решить некоторые обнаруженные проблемы.

Я начну эту главу с обзора нескольких представлений каталога SQL Server и покажу, как выявлять ряд проблем проектирования базы данных. Затем продемонстрирую, как анализировать использование индекса и его операционную статистику, чтобы обнаружить неэффективное индексирование, а также предложу несколько сценариев для этого анализа, в том числе код, который позволит просмотреть несколько показателей индекса в одной сводке.

Анализ схемы базы данных

В SQL Server есть довольно много *представлений каталога*[1], которые отображают информацию об объектах уровня сервера и базы данных. Они чрезвычайно полезны, когда нужно проанализировать и обнаружить дефекты в схеме базы данных.

На рис. 14.1 показано несколько представлений каталога, связанных с объектами базы данных, включая их зависимости и ключевые атрибуты. Это лишь малая

[1] https://oreil.ly/BIl1D

часть доступных представлений, по которой можно судить, сколько информации такого рода предоставляет SQL Server.

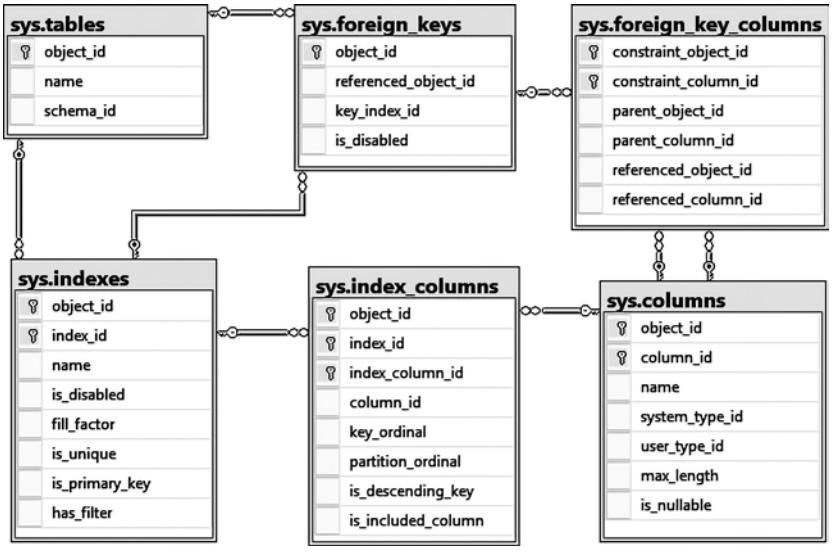


Рис. 14.1. Представления каталога

С помощью этих представлений можно обнаружить несколько распространенных проблем проектирования баз данных, например:

- таблицы-кучи;
- индексы на столбцах `uniqueidentifier`;
- широкие и неуникальные кластеризованные индексы;
- недоверенные внешние ключи;
- неиндексированные внешние ключи;
- избыточные индексы;
- высокие значения идентификаторов.

Рассмотрим эти проблемы по порядку.

Таблицы-кучи

Таблицы-кучи — это неоднозначная тема. У них есть своя ниша: например, они полезны в некоторых процессах ETL (извлечение, преобразование, загрузка), когда критична скорость вставки и нет возможности использовать таблицы,

оптимизированные для памяти. Но, как упоминалось в главе 5, лучше избегать таблиц-куч, потому что таблицы с кластеризованными индексами работают эффективнее на большинстве рабочих нагрузок.

Встретив таблицы-кучи, я обычно изучаю их и пробую создать для них кластеризованные индексы. Если это не удается, я ищу неэффективные таблицы-кучи и перестраиваю их.

Код в листинге 14.1 выводит список таблиц-куч в базе данных. Имеет смысл просмотреть статистику использования индексов в этих таблицах, чтобы понять, есть ли естественный кандидат на роль кластеризованного индекса. (Я расскажу об этом позже в этой главе.)

Если вы запускаете этот и другие сценарии из этой главы в версиях SQL Server до 2014, удалите предикаты из столбца `sys.tables.is_memory_optimized`.

Листинг 14.1. Получение информации о таблицах-кучах в базе данных

```
SELECT
    t.object_id
    ,s.name + '.' + t.name AS [table]
    ,p.rows
FROM
    sys.tables t WITH (NOLOCK)
        JOIN sys.schemas s WITH (NOLOCK) ON
            t.schema_id = s.schema_id
        CROSS APPLY
        (
            SELECT SUM(p.rows) AS [rows]
            FROM sys.partitions p WITH (NOLOCK)
            WHERE t.object_id = p.object_id AND p.index_id = 0
        ) p
WHERE
    t.is_memory_optimized = 0 AND
    t.is_ms_shipped = 0 AND
    EXISTS
    (
        SELECT *
        FROM sys.indexes i WITH (NOLOCK)
        WHERE t.object_id = i.object_id AND i.index_id = 0
    )
ORDER BY
    p.rows DESC
OPTION (RECOMPILE, MAXDOP 1);
```

Для таблиц-куч стоит отслеживать две основные метрики. Во-первых, это количество *переадресованных записей*. В отличие от индексов на основе B-деревьев, SQL Server не разбивает страницу таблицы-кучи, когда на этой странице не

хватает места, чтобы разместить новую версию строки. Вместо этого он помещает новую строку на другую страницу и заменяет исходную строку небольшой структурой, которая называется *указателем переадресации*. От большого количества указателей переадресации и переадресованных записей снижается производительность операций ввода/вывода в таблице, потому что SQL Server приходится выполнять дополнительные операции чтения для доступа к данным.

Вторая метрика — *внутренняя фрагментация*, то есть количество свободного места, доступного на страницах данных. Высокая степень внутренней фрагментации увеличивает размер таблицы на диске и в памяти, что плохо сказывается на производительности.

В листинге 14.2 показано, как обнаруживать неэффективные таблицы-кучи. В коде используется функция `sys.dm_db_index_physical_stats`[1], выполняющаяся в режиме `DETAILED`. В этом режиме функция просматривает всю таблицу, поэтому ее лучше запускать не в промышленной среде, а, например, для недавней резервной копии базы данных, восстановленной на вспомогательном сервере.

Чем больше в таблице указателей переадресации и внутренней фрагментации, тем сильнее эти факторы влияют на производительность, хотя трудно указать точные пороги, при которых они вызывают проблемы. Как правило, количество указателей переадресации не должно превышать 2–3 % от общего числа строк в таблице. Для внутренней фрагментации мне кажутся приемлемыми значения в пределах от 20 до 25 %. В больших таблицах я предпочитаю еще более низкие значения, особенно когда существенная часть данных становится статической.

Обнаружив неэффективные таблицы-кучи, их можно перестроить с помощью инструкции `ALTER TABLE REBUILD`.

Листинг 14.2. Обнаружение неэффективных таблиц-куч

```
SELECT TOP 25
    t.object_id
    ,s.name + '.' + t.name AS [table]
    ,SUM(ips.record_count) AS [rows]
    ,SUM(ips.forwarded_record_count)
            AS [forwarding pointers]
    ,SUM(ips.avg_page_space_used_in_percent * ips.page_count) /
        NULLIF(SUM(ips.page_count),0)
            AS [internal fragmentation %]
FROM
    sys.tables t WITH (NOLOCK)
        JOIN sys.schemas s WITH (NOLOCK) ON
            t.schema_id = s.schema_id
        CROSS APPLY
            sys.dm_db_index_physical_stats
                (DB_ID(),t.object_id,0,NULL,'DETAILED') ips
```

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

```
WHERE
    t.is_memory_optimized = 0 AND
    t.is_ms_shipped = 0 AND
    EXISTS
    (
        SELECT *
        FROM sys.indexes i WITH (NOLOCK)
        WHERE t.object_id = i.object_id AND i.index_id = 0
    )
GROUP BY
    t.object_id, s.name, t.name
ORDER BY
    [forwarding pointers] DESC
OPTION (RECOMPILE, MAXDOP 1);
```

Индексы с типом данных uniqueidentifier

Индексы на основе B-деревьев, где в качестве ключей используются случайно сгенерированные значения, часто создают проблемы с производительностью. Случайные значения могут вызвать существенную фрагментацию индекса и медленно работают при больших пакетных операциях.

Для генерации случайных значений очень часто используют тип `uniqueidentifier`. Код в листинге 14.3 показывает индексы, у которых самый левый ключевой столбец имеет этот тип. Обнаружив такие индексы, можно также проанализировать их фрагментацию с помощью представления `sys.dm_db_index_physical_stats`.

Листинг 14.3. Получение индексов с типом `uniqueidentifier` в крайнем левом столбце

```
SELECT
    t.object_id
    ,s.name + '.' + t.name AS [table]
    ,i.name AS [index]
    ,i.is_disabled
    ,p.rows
FROM
    sys.tables t WITH (NOLOCK)
        JOIN sys.schemas s WITH (NOLOCK) ON
            t.schema_id = s.schema_id
        JOIN sys.indexes i WITH (NOLOCK) ON
            t.object_id = i.object_id
        CROSS APPLY
        (
            SELECT SUM(p.rows) AS [rows]
            FROM sys.partitions p WITH (NOLOCK)
            WHERE i.object_id = p.object_id AND i.index_id = p.index_id
        ) p
WHERE
    t.is_memory_optimized = 0 AND
    i.type in (1,2) AND /* кластеризованные и некластеризованные индексы */
```

```
    i.is_hypothetical = 0 AND
    EXISTS
    (
        SELECT *
        FROM
            sys.index_columns ic WITH (NOLOCK)
                JOIN sys.columns c WITH (NOLOCK) ON
                    ic.object_id = c.object_id AND
                    ic.column_id = c.column_id
        WHERE
            ic.object_id = i.object_id AND
            ic.index_id = i.index_id AND
            ic.key_ordinal = 1 AND
            c.system_type_id = 36 /* uniqueidentifier */
    )
ORDER BY
    p.[rows] DESC
OPTION (RECOMPILE, MAXDOP 1);
```

Стоит разобраться, как генерируются новые значения `uniqueidentifier`. Случайные значения, сгенерированные в приложении или с помощью функции `NEWID()`, вызывают проблемы, а вот функция `NEWSEQUENTIALID()` генерирует постоянно увеличивающиеся значения `GUID`, которые ведут себя аналогично столбцам идентификаторов (за исключением того, что они менее эффективны из-за большего размера типа данных). Имейте в виду, что функция `NEWSEQUENTIALID()` может генерировать значения меньше предыдущих после перезапуска ОС или аварийного переключения.

К сожалению, нет простого способа решить проблемы, связанные с индексированными случайными ключами. Я рекомендую проанализировать степень фрагментации, наладить стратегию обслуживания индекса и отрегулировать свойство индекса `FILLFACTOR`, чтобы минимизировать разбиение страниц, а также попробовать провести рефакторинг кода, когда это возможно.

Широкие и неуникальные кластеризованные индексы

Для хорошей производительности системы важны эффективные кластеризованные индексы. Помимо поддержки критических запросов, кластеризованный индекс в идеале обладает следующими тремя характеристиками:

Статичность

Во-первых, идеальный кластеризованный индекс должен быть *статическим*. Обычно следует избегать обновления кластеризованных ключей, потому что это очень затратная операция: она требует перемещать строки данных в другое место в B-дереве (кластеризованного индекса), а затем обновлять идентификаторы строк во всех строках некластеризованного индекса, которые ссылаются на измененные данные.

Узость

Во-вторых, идеальный кластеризованный индекс должен быть *узким*. Поскольку ключевые столбцы кластеризованного индекса представлены как идентификаторы строк во всех некластеризованных индексах, широкие ключи кластеризованного индекса порождают широкие и, следовательно, менее эффективные некластеризованные индексы.

Нельзя точно сказать, насколько узким должен быть индекс, потому что это зависит от многих факторов. Просто помните, что узкие индексы более эффективны, чем широкие. Например, первичные ключи, представленные как 4-байтовые значения типа `int` или 8-байтовые `bigint`, всегда эффективнее, чем 16-байтовые `uniqueidentifier`.

Уникальность

Наконец, в-третьих, идеальный кластеризованный индекс должен быть определен как *уникальный* (*unique*). Если он не будет таковым, SQL Server добавит еще один внутренний 4-байтовый столбец с именем `uniquifier` (не путать с `uniqueidentifier`!), который обеспечивает уникальность ключей кластеризованного индекса. Этот столбец увеличивает размер ключей кластеризованных и некластеризованных индексов, так что его лучше избегать.

В листинге 14.4 показаны два запроса. Первый запрос выводит 25 таблиц с самыми широкими ключами кластеризованного индекса на основе типов данных ключевого столбца. Второй запрос выдает список таблиц с неуникальными кластеризованными индексами.

Листинг 14.4. Обнаружение неэффективных кластеризованных индексов

```
SELECT TOP 25
    t.object_id
    ,s.name + '.' + t.name AS [table]
    ,p.rows
    ,ic.[max length]
FROM
    sys.tables t WITH (NOLOCK)
        JOIN sys.schemas s WITH (NOLOCK) ON
            t.schema_id = s.schema_id
        CROSS APPLY
        (
            SELECT SUM(p.rows) AS [rows]
            FROM sys.partitions p WITH (NOLOCK)
            WHERE t.object_id = p.object_id AND p.index_id = 1
        ) p
        CROSS APPLY
        (
            SELECT SUM(c.max_length) as [max length]
            FROM
```

```
                sys.indexes i
                    JOIN sys.index_columns ic WITH (NOLOCK) ON
                        i.object_id = ic.object_id AND
                        i.index_id = ic.index_id AND
                        ic.is_included_column = 0
                    JOIN sys.columns c WITH (NOLOCK) ON
                        ic.object_id = c.object_id AND
                        ic.column_id = c.column_id
            WHERE
                i.object_id = t.object_id AND
                i.index_id = 1 AND
                i.type = 1
        ) ic
WHERE
    t.is_memory_optimized = 0
ORDER BY
    ic.[max length] DESC
OPTION (RECOMPILE, MAXDOP 1);

-- Неуникальные кластеризованные индексы
SELECT
    t.object_id
    ,s.name + ,.' + t.name AS [table]
    ,p.rows
FROM
    sys.tables t WITH (NOLOCK)
        JOIN sys.schemas s WITH (NOLOCK) ON
            t.schema_id = s.schema_id
        CROSS APPLY
        (
            SELECT SUM(p.rows) AS [rows]
            FROM sys.partitions p WITH (NOLOCK)
            WHERE t.object_id = p.object_id AND p.index_id = 1
        ) p
WHERE
    t.is_memory_optimized = 0 AND
    EXISTS
    (
        SELECT *
        FROM sys.indexes i WITH (NOLOCK)
        WHERE
            t.object_id = i.object_id AND
            i.index_id = 1 AND
            i.is_unique = 0 AND
            i.type = 1 /* кластеризованный индекс */
    )
ORDER BY
    p.[rows] DESC
OPTION (RECOMPILE, MAXDOP 1);
```

Не делайте поспешных выводов исключительно по результатам этих запросов. Хорошие кластеризованные индексы, поддерживающие критические запросы, могут стать преимуществом, которое с лихвой компенсирует накладные расходы,

связанные с широкими и неуникальными индексами. Прежде чем принимать решение о рефакторинге схемы базы данных, изучите запросы и индексирование в целом.

Тем не менее всегда лучше делать и кластеризованные, и некластеризованные индексы уникальными, если данные в ключе индекса уникальны. В этом случае кластеризованный индекс становится эффективнее, потому что нет лишних накладных расходов на столбец `uniquifier`, а оптимизатор запросов может создавать более эффективные планы выполнения.

Недоверенные внешние ключи

За исключением редких особых случаев, всегда стоит определить ограничения внешних ключей в базе данных. Они улучшают качество данных и снижают количество ошибок. Более того, они часто повышают производительность: например, при наличии внешних ключей оптимизатор запросов может удалить ненужные соединения между таблицами.

Создать ограничения внешних ключей можно двумя способами. По умолчанию они создаются как *доверенные (trusted)*, и SQL Server проверяет, чтобы существующие данные в таблицах не нарушали это ограничение. Можно задать ограничение как *недоверенное (untrusted)*, используя предложение `WITH NOCHECK` в инструкции `ALTER TABLE`. В таком случае SQL Server применяет ограничение с этого момента, но не проверяет уже существующие данные.

К сожалению, недоверенные ограничения внешних ключей сужают возможности оптимизатора запросов, поэтому такие ключи лучше проверять. Листинг 14.5 позволяет обнаруживать недоверенные ограничения в базе данных.

Листинг 14.5. Выбор недоверенных ограничений внешних ключей

```
SELECT
    fk.is_disabled
    ,fk.is_not_trusted
    ,fk.name AS [FK]
    ,ps.name + '.' + pt.name AS [Referencing Table / Detail]
    ,rs.name + '.' + rt.name AS [Referenced Table / Master]
    ,fk.update_referential_action_desc
    ,fk.delete_referential_action_desc
FROM
    sys.foreign_keys fk WITH (NOLOCK)
        JOIN sys.tables pt WITH (NOLOCK) ON
            fk.parent_object_id = pt.object_id
        JOIN sys.schemas ps WITH (NOLOCK) ON
            pt.schema_id = ps.schema_id
        JOIN sys.tables rt WITH (NOLOCK) ON
            fk.referenced_object_id = rt.object_id
        JOIN sys.schemas rs WITH (NOLOCK) ON
```

```
            rt.schema_id = rs.schema_id
WHERE
    fk.is_not_trusted = 1 OR fk.is_disabled = 1
OPTION (RECOMPILE, MAXDOP 1);
```

Проверить ограничения можно с помощью команды `ALTER TABLE WITH CHECK CHECK CONSTRAINT` (ключевое слово `CHECK` указывается в инструкции дважды). Но должен предупредить: эта операция просматривает таблицу, устанавливая блокировку модификации схемы (`Sch-M`) на время просмотра. Планируйте эту операцию на время простоя, особенно если работаете с большими таблицами.

Неиндексированные внешние ключи

Ограничения внешнего ключа требуют, чтобы в главной таблице был индекс для столбцов ограничений. Однако в подчиненной таблице (которая ссылается на главную) индекс необязателен, что может привести к серьезным проблемам при проверке ссылочной целостности.

Предположим, вы удаляете строку в главной таблице. SQL Server должен проверить, не нарушила ли эта операция ограничение, и, возможно, выполнить каскадные действия в подчиненной таблице. Без индекса это приведет к просмотру подчиненной таблицы, а в результате — к снижению производительности и потенциальному блокированию.

Код из листинга 14.6 возвращает список ограничений внешнего ключа, для которых не определено соответствующих индексов в подчиненных таблицах. За некоторыми исключениями, в таблицах нужно создавать индексы, чтобы облегчить операции ссылочной целостности.

Очевидно, стоит учитывать, как планируется использовать индекс. Например, если внешние ключи ссылаются на статические таблицы подстановок, которые никогда не обновляются и не удаляются, то индексы могут не понадобиться. В то же время если таблицы, на которые ведут ссылки, не являются статическими и в них ожидаются проверки ссылочной целостности или каскадные операции, создайте индексы для поддержки этих операций.

Еще один фактор — это запросы, которые объединяют таблицы, фильтруя данные по атрибутам таблицы подстановок. Если в ссылающихся таблицах есть индексы, это поможет создать более эффективные планы выполнения.

Листинг 14.6. Получение неиндексированных ограничений внешнего ключа

```
SELECT
    fk.is_disabled
    ,fk.is_not_trusted
    ,fk.name as [FK]
    ,ps.name + '.' + pt.name AS [Referencing Table / Detail]
```

```
    ,rs.name + '.' + rt.name AS [Referenced Table / Master]
    ,fk.update_referential_action_desc
    ,fk.delete_referential_action_desc
    ,fk_cols.cols as [fk columns]
FROM
    sys.foreign_keys fk WITH (NOLOCK)
        JOIN sys.tables pt WITH (NOLOCK) ON
            fk.parent_object_id = pt.object_id
        JOIN sys.schemas ps WITH (NOLOCK) ON
            pt.schema_id = ps.schema_id
        JOIN sys.tables rt WITH (NOLOCK) ON
            fk.referenced_object_id = rt.object_id
        JOIN sys.schemas rs WITH (NOLOCK) ON
            rt.schema_id = rs.schema_id
        CROSS APPLY
        (
            SELECT
                (
                    SELECT
                        UPPER(col.name) AS [text()]
                        ,',' AS [text()]
                    FROM
                        sys.foreign_key_columns fkc WITH (NOLOCK)
                            JOIN sys.columns col WITH (NOLOCK) ON
                                fkc.parent_object_id = col.object_id AND
                                fkc.parent_column_id = col.column_id
                    WHERE
                        fkc.constraint_object_id = fk.object_id
                    ORDER BY
                        fkc.constraint_column_id
                    FOR XML PATH(,')
                ) as cols
        ) fk_cols
WHERE
    NOT EXISTS
    (
        SELECT *
        FROM
            sys.indexes i WITH (NOLOCK)
                CROSS APPLY
                (
                    SELECT
                        (
                            SELECT
                                UPPER(col.name) AS [text()]
                                ,',' AS [text()]
                            FROM
                                sys.index_columns ic WITH (NOLOCK)
                                    JOIN sys.columns col WITH (NOLOCK) ON
                                        ic.object_id = col.object_id AND
                                        ic.column_id = col.column_id
                            WHERE
                                i.object_id = ic.object_id AND
```

```
                                i.index_id = ic.index_id AND
                                ic.is_included_column = 0
                            ORDER BY
                                ic.partition_ordinal
                            FOR XML PATH(',')
                        ) AS cols
                ) idx_col
        WHERE
            i.object_id = fk.parent_object_id AND
            CHARINDEX(fk_cols.cols,idx_col.cols) = 1 AND
            i.is_disabled = 0 AND
            i.is_hypothetical = 0 AND
            i.has_filter = 0 AND
            i.type IN (1,2)
    )
ORDER BY
    [Referenced Table / Master]
OPTION (RECOMPILE, MAXDOP 1);
```

Этот сценарий исключает из расчетов отфильтрованные индексы, из-за чего в выходных данных могут по ошибке оказаться отдельные внешние ключи. Представьте, что в подчиненной таблице есть обнуляемый столбец `Col` и индекс, определенный с помощью фильтра `Col IS NOT NULL`. В сценарии этот индекс будет проигнорирован, несмотря на то что он отлично годится для поддержки ссылочной целостности. При анализе обратите внимание на эту особенность.

Избыточные индексы

Как вы помните из главы 5, данные в составных индексах на основе B-деревьев сортируются, начиная с крайнего левого ключевого столбца и заканчивая крайним правым. SQL Server может использовать индексы для операции поиска по индексу, если в крайних левых столбцах есть предикаты, поддерживающие поиск и аргументы (SARG).

В качестве примера рассмотрим два индекса: `IDX1(LastName)` и `IDX2(LastName, FirstName)`. Данные в обоих индексах сортируются по `LastName`. В `IDX2` есть данные, которые сортируются по `LastName` внутри каждого `FirstName`, а в `IDX1` по `LastName` ничего не сортируется. Оба индекса поддерживают поиск по индексу по `FirstName`, что делает `IDX1` избыточным и ненужным.

В некоторых особых случаях избыточные индексы с узкими ключами могут пригодиться для операций быстрого просмотра индекса. Однако такие ситуации встречаются редко.

В листинге 14.7 приведен код, с помощью которого можно обнаруживать потенциально избыточные индексы с соответствующими крайними левыми

столбцами. Используйте этот сценарий с осторожностью, потому что часто встречаются составные индексы с общим крайним левым столбцом.

Характерный пример этой ситуации — многопользовательская база данных, в которой хранятся данные от нескольких клиентов или учетных записей. Предложения `WHERE` в запросах к этой базе обычно содержат столбец `CustomerId` или `AccountId`, и часто это крайний левый столбец в большинстве индексов.

Листинг 14.7. Поиск потенциально избыточных индексов

```
SELECT
    s.name + '.' + t.name AS [Table]
    ,i1.index_id AS [I1 ID]
    ,i1.name AS [I1 Name]
    ,dupIdx.index_id AS [I2 ID]
    ,dupIdx.name AS [I2 Name]
    ,LEFT(i1_col.key_col,LEN(i1_col.key_col) - 1) AS [I1 Keys]
    ,LEFT(i1_col.included_col,LEN(i1_col.included_col) - 1) AS [I1 Included Col]
    ,i1.filter_definition AS [I1 Filter]
    ,LEFT(i2_col.key_col,LEN(i2_col.key_col) - 1) AS [I2 Keys]
    ,LEFT(i2_col.included_col,LEN(i2_col.included_col) - 1) AS [I2 Included Col]
    ,dupIdx.filter_definition AS [I2 Filter]
    ,IIF(
        CHARINDEX(i1_col.key_col, i2_col.key_col) = 1 OR
        CHARINDEX(i2_col.key_col, i1_col.key_col) = 1,'Yes','No'
    ) AS [Fully Redundant]
FROM
    sys.tables t WITH (NOLOCK)
        JOIN sys.indexes i1 wITH (NOLOCK) ON
            t.object_id = i1.object_id
        JOIN sys.index_columns ic1 WITH (NOLOCK) ON
            ic1.object_id = i1.object_id AND
            ic1.index_id = i1.index_id AND
            ic1.key_ordinal = 1
        JOIN sys.columns c WITH (NOLOCK) ON
            c.object_id = ic1.object_id AND
            c.column_id = ic1.column_id
        JOIN sys.schemas s WITH (NOLOCK) ON
            t.schema_id = s.schema_id
        CROSS APPLY
        (
            SELECT i2.index_id, i2.name, i2.filter_definition
            FROM
                sys.indexes i2 WITH (NOLOCK)
                    JOIN sys.index_columns ic2 WITH (NOLOCK) ON
                        ic2.object_id = i2.object_id AND
                        ic2.index_id = i2.index_id AND
                        ic2.key_ordinal = 1
            WHERE
                i2.object_id = i1.object_id AND
                i2.index_id > i1.index_id AND
                ic2.column_id = ic1.column_id AND
```

```
        i2.type in (1,2) AND
        i2.is_disabled = 0 AND
        i2.is_hypothetical = 0 AND
        (
            i1.has_filter = i2.has_filter AND
            ISNULL(i1.filter_definition,'') =
                ISNULL(i2.filter_definition,'')
        )
) dupIdx
CROSS APPLY
(
    SELECT
        (
            SELECT
                col.name AS [text()]
                ,IIF(icol_meta.is_descending_key = 1, , DESC','')
                    AS [text()]
                ,',' AS [text()]
            FROM
                sys.index_columns icol_meta WITH (NOLOCK)
                    JOIN sys.columns col WITH (NOLOCK) ON
                        icol_meta.object_id = col.object_id AND
                        icol_meta.column_id = col.column_id
            WHERE
                icol_meta.object_id = i1.object_id AND
                icol_meta.index_id = i1.index_id AND
                icol_meta.is_included_column = 0
            ORDER BY
                icol_meta.key_ordinal
            FOR XML PATH(,')
        ) AS key_col
        ,(
            SELECT
                col.name AS [text()]
                ,',' AS [text()]
            FROM
                sys.index_columns icol_meta WITH (NOLOCK)
                    JOIN sys.columns col WITH (NOLOCK) ON
                        icol_meta.object_id = col.object_id AND
                        icol_meta.column_id = col.column_id
            WHERE
                icol_meta.object_id = i1.object_id AND
                icol_meta.index_id = i1.index_id AND
                icol_meta.is_included_column = 1
            ORDER BY
                col.name
            FOR XML PATH(,')
        ) AS included_col
) i1_col
CROSS APPLY
(
    SELECT
        (
```

```
                    SELECT
                        col.name AS [text()]
                        ,IIF(icol_meta.is_descending_key = 1, , DESC','')
                            AS [text()]
                        ,',' AS [text()]
                    FROM
                        sys.index_columns icol_meta WITH (NOLOCK)
                            JOIN sys.columns col WITH (NOLOCK) ON
                                icol_meta.object_id = col.object_id AND
                                icol_meta.column_id = col.column_id
                    WHERE
                        icol_meta.object_id = t.object_id AND
                        icol_meta.index_id = dupIdx.index_id AND
                        icol_meta.is_included_column = 0
                    ORDER BY
                        icol_meta.key_ordinal
                    FOR XML PATH(,')
                ) AS key_col
                ,(
                    SELECT
                        col.name AS [text()]
                        ,',' AS [text()]
                    FROM
                        sys.index_columns icol_meta WITH (NOLOCK)
                            JOIN sys.columns col WITH (NOLOCK) ON
                                icol_meta.object_id = col.object_id AND
                                icol_meta.column_id = col.column_id
                    WHERE
                        icol_meta.object_id = t.object_id AND
                        icol_meta.index_id = dupIdx.index_id AND
                        icol_meta.is_included_column = 1
                    ORDER BY
                        col.name
                    FOR XML PATH(,')
                ) AS included_col
        ) i2_col
WHERE
    i1.is_disabled = 0 AND
    i1.is_hypothetical = 0 AND
    i1.type in (1,2)
ORDER BY
    s.name, t.name, i1.index_id
OPTION (RECOMPILE, MAXDOP 1);
```

Обычно можно удалить полностью избыточные индексы (обратите внимание на фильтры индексов и неключевые столбцы) и дополнительно поискать кандидатов для консолидации индексов. Например, индексы `IDX3` и `IDX4`, определенные в листинге 14.8, можно консолидировать в `IDX5`. Обращайте внимание на статистику использования индексов (я расскажу об этом позже в этой главе), потому что она содержит сведения о том, какие индексы используются редко.

Листинг 14.8. Примеры консолидации индексов

```
CREATE INDEX IDX3 ON T(LastName, FirstName) INCLUDE(Phone);
CREATE INDEX IDX4 ON T(LastName) INCLUDE (SSN);
--IDX3 и IDX4 можно консолидировать в IDX5
CREATE INDEX IDX5  ON T(LastName, FirstName) INCLUDE(Phone,SSN);
```

Прежде чем удалять или отключать индексы, проанализируйте схему базы данных и запросы: убедитесь, что эти индексы не упоминаются в индексных указаниях!

Высокие значения идентификаторов

Существует дурацкая, но в то же время опасная ситуация, с которой вы можете столкнуться: исчерпание емкости целочисленных столбцов. Представьте таблицу с первичным ключом `INT IDENTITY PRIMARY KEY`, который достигает 2 147 483 647, то есть максимального значения для типа данных `INT`. Все дальнейшие операции `INSERT` в таблице завершатся ошибкой, что может привести к аварии на промышленном сервере.

Что еще хуже, эту ситуацию сложно исправить. Можно изменить тип данных с `INT` на `BIGINT`, но выполнение инструкции `ALTER TABLE ALTER COLUMN` для больших таблиц может занять часы или даже дни. Более того, на время этой операции таблица окажется под блокировкой модификации схемы (`Sch-M`). К сожалению, я не раз видел, как люди теряли работу, когда происходила такая беда.

Код из листинга 14.9 ищет столбцы идентификаторов и последовательности, у которых вскоре исчерпается мощность. Регулярно запускайте этот сценарий в своих системах и заблаговременно устраняйте риски, если видите проблемы.

Листинг 14.9. Поиск столбцов идентификаторов с заканчивающейся емкостью

```
DECLARE
    @Types TABLE
    (
        type_id INT NOT NULL PRIMARY KEY,
        name SYSNAME NOT NULL,
        max_val DECIMAL(38) NOT NULL
    )
INSERT INTO @Types(type_id, name, max_val)

VALUES
    (48,'TINYINT',255)
    ,(52,'SMALLINT',32767)
    ,(56,'INT',2147483647)
    ,(127,'BIGINT',9223372036854775807)
    ,(108,'NUMERIC',99999999999999999999999999999999999999) -- 10^38-1
    ,(106,'DECIMAL',99999999999999999999999999999999999999); -- 10^38-1
```

```
DECLARE
    @percentThreshold INT = 50;

;WITH CTE
AS
(
    SELECT
        s.name + '.' + t.name AS [table]
        ,c.name AS [column]
        ,tp.name + IIF(tp.type_id IN (106,108), '(' +
            CONVERT(VARCHAR(2),c.precision) + ')','') AS [type]
        ,CONVERT(DECIMAL(38),IDENT_CURRENT(t.name)) AS [identity]
        ,CASE
            WHEN tp.type_id IN (106,108)
            THEN
                CASE
                    WHEN c.precision < 38
                    THEN POWER(CONVERT(DECIMAL(38),10),c.precision) - 1
                    ELSE tp.max_val
                END
            ELSE
                tp.max_val
        END AS [max value]
    FROM
        sys.tables t WITH (NOLOCK)
            JOIN sys.schemas s WITH (NOLOCK) ON
                t.schema_id = s.schema_id
            JOIN sys.columns c WITH (NOLOCK) ON
                c.object_id = t.object_id
            JOIN @Types tp ON
                tp.type_id = c.system_type_id
    WHERE
        c.is_identity = 1
)
SELECT
    *
    ,CONVERT(DECIMAL(6,3),[identity] / [max value] * 100.)
        AS [percent full]
FROM
    CTE
WHERE
    CONVERT(DECIMAL(6,3),[identity] / [max value] * 100.) >
        @percentThreshold
ORDER BY
    [percent full] DESC;

-- Sequences
;WITH CTE
AS
(
    SELECT
        s.name + ,.' + seq.name AS [sequence]
        ,tp.name AS [type]
```

```
        ,CASE tp.type_id
            WHEN 48 THEN
                CONVERT(DECIMAL(38),CONVERT(TINYINT,seq.current_value))
            WHEN 52 THEN
                CONVERT(DECIMAL(38),CONVERT(SMALLINT,seq.current_value))
            WHEN 56 THEN
                CONVERT(DECIMAL(38),CONVERT(INT,seq.current_value))
            WHEN 127 THEN
                CONVERT(DECIMAL(38),CONVERT(BIGINT,seq.current_value))
            WHEN 106 THEN
                CONVERT(DECIMAL(38),seq.current_value)
            WHEN 108 THEN
                CONVERT(DECIMAL(38),seq.current_value)
        END as [current]
        ,CASE tp.type_id
            WHEN 48 THEN
                CONVERT(DECIMAL(38),CONVERT(TINYINT,seq.maximum_value))
            WHEN 52 THEN
                CONVERT(DECIMAL(38),CONVERT(SMALLINT,seq.maximum_value))
            WHEN 56 THEN
                CONVERT(DECIMAL(38),CONVERT(INT,seq.maximum_value))
            WHEN 127 THEN
                CONVERT(DECIMAL(38),CONVERT(BIGINT,seq.maximum_value))
            WHEN 106 THEN
                CONVERT(DECIMAL(38),seq.maximum_value)
            WHEN 108 THEN
                CONVERT(DECIMAL(38),seq.maximum_value)
        END as [max value]
    FROM
        sys.sequences seq WITH (NOLOCK)
            JOIN sys.schemas s WITH (NOLOCK) ON
                seq.schema_id = s.schema_id
            JOIN @Types tp ON
                tp.type_id = seq.system_type_id
)
SELECT
    *
    ,CONVERT(DECIMAL(6,3), [current] / [max value] * 100.) as [percent full]
FROM
    CTE
WHERE
    CONVERT(DECIMAL(6,3), [current] / [max value] * 100.) >
        @percentThreshold
ORDER BY
    [percent full] DESC;
```

Очевидно, этот сценарий не выявит ситуацию, когда значения генерируются в коде. Он также не проверяет возможные риски нехватки емкости, которые не связаны с идентификаторами и последовательностями. Я включил в сопутствующие материалы этой книги сценарий (созданный Эрландом Соммарскогом, который проделал большую работу по рецензированию книги!), показывающий максимальные значения ключей индексов во всех индексах базы данных.

Мы рассмотрели лишь несколько примеров того, как с помощью представлений каталога можно выявлять очаги неэффективности в схемах баз данных. Исследуйте другие представления и не забывайте проявлять творческий подход!

Теперь рассмотрим еще один важнейший источник информации: статистику использования индекса и операций.

Анализ индекса

Всем известно, что индексы помогают повысить производительность запросов. Но все имеет свою цену: индексы увеличивают объем данных в базе, потребляют дополнительную память и добавляют накладные расходы при модификации данных. От большого количества неэффективных и/или неиспользуемых индексов производительность системы может значительно снизиться.

Оптимизация запросов — это важная часть настройки производительности, в ходе которой в базе данных часто создаются новые индексы. Однако, когда вы работаете не в аварийной ситуации, я всегда рекомендую уделять время анализу и удалению неэффективных индексов *перед* тем, как создавать новые. Это устранит их накладные расходы и упростит процесс оптимизации.

В SQL Server есть два динамических административных представления для анализа использования индексов. Первое, `sys.dm_db_index_usage_stats`[1], показывает количество запросов, которые использовали индекс для операций поиска, просмотра, обновления и уточнения. Второе представление, `sys.dm_db_index_operational_stats`[2], дает информацию о доступе на уровне строк и операционные метрики, включая показатели ввода/вывода, сведения о блокировках и некоторые другие данные.

Способы сбора этих метрик существенно различаются. Представление `sys.dm_db_index_usage_stats` обеспечивает информацию на уровне запросов, а `sys.dm_db_index_operational_stats` работает на уровне строк. Например, если запрос вставляет в таблицу 1000 строк, то в первом представлении число `user_updates` увеличится на 1, а во втором представлении столбец `leaf_insert_count` вырастет на 1000.

SQL Server не сохраняет статистику использования индекса. Метрики в этих представлениях очищаются при перезапуске SQL Server или отключении базы данных. Более того, в некоторых старых сборках SQL Server (SQL Server 2012 до SP2 CU12 и SP3 CU3; SQL Server 2014 до SP2) метрики очищаются, когда выполняется операция перестроения индекса. Это нужно учитывать в ходе анализа: убеждайтесь, что статистика репрезентативна и не пропускает важные, но редко

[1] https://oreil.ly/IcA15

[2] https://oreil.ly/AWvEX

выполняемые запросы, такие как, например, критический процесс в банковской системе, который запускается по расписанию один раз в месяц.

Также следует изучить использование индекса на доступных для чтения вторичных репликах в группах доступности. Нередко бывает, что отдельные индексы поддерживают запросы на вторичных узлах, а на первичном узле эти индексы могут отображаться как неиспользуемые.

Давайте рассмотрим оба представления подробнее и разберемся, как интерпретировать полученную из них информацию.

Представление sys.dm_db_index_usage_stats

Представление `sys.dm_db_index_usage_stats` — один из основных инструментов анализа использования индекса. Оно содержит информацию о том, как часто используется индекс или, если быть точнее, сколько раз запросы задействовали индекс и он появлялся в планах выполнения.

Данные группируются по методам доступа, и вы увидите отдельные метрики для операций поиска (seek), просмотра (scan) и уточнения (lookup). Наконец, в представлении также показано, как часто индекс обновлялся, что помогает оценить связанные с ним затраты на обновление.

В листинге 14.10 показан код, использующий это представление. Я переименовал некоторые выходные столбцы, чтобы сделать их компактнее. В этой главе я буду попеременно использовать и имена столбцов представления, и мои псевдонимы.

Листинг 14.10. Использование представления `sys.dm_db_index_usage_stats`

```
SELECT
    t.object_id
    ,i.index_id
    ,s.name + '.' + t.name AS [Table]
    ,i.name AS [Index]
    ,i.type_desc
    ,i.has_filter AS [Filtered]
    ,i.is_unique AS [Unique]
    ,p.rows AS [Rows]
    ,ius.user_seeks AS [Seeks]
    ,ius.user_scans AS [Scans]
    ,ius.user_lookups AS [Lookups]
    ,ius.user_seeks + ius.user_scans + ius.user_lookups AS [Reads]
    ,ius.user_updates AS [Updates]
    ,ius.last_user_seek AS [Last Seek]
    ,ius.last_user_scan AS [Last Scan]
    ,ius.last_user_lookup AS [Last Lookup]
    ,ius.last_user_update AS [Last Update]
FROM
    sys.tables t WITH (NOLOCK)
        JOIN sys.indexes i WITH (NOLOCK) ON
```

```
            t.object_id = i.object_id
        JOIN sys.schemas s WITH (NOLOCK) ON
            t.schema_id = s.schema_id
    CROSS APPLY
    (
        SELECT SUM(p.rows) AS [rows]
        FROM sys.partitions p WITH (NOLOCK)
        WHERE
            i.object_id = p.object_id AND
            i.index_id = p.index_id
    ) p
    LEFT OUTER JOIN sys.dm_db_index_usage_stats ius ON
        ius.database_id = DB_ID() AND
        ius.object_id = i.object_id AND
        ius.index_id = i.index_id
WHERE
    i.is_disabled = 0 AND
    i.is_hypothetical = 0 AND
    t.is_memory_optimized = 0 AND
    t.is_ms_shipped = 0
ORDER BY
    s.name, t.name, i.index_id
OPTION (RECOMPILE, MAXDOP 1);
```

На рис. 14.2 показан вывод кода.

	object_id	index_id	Table	Index	type_desc	filtered	Unique	Rows	Seeks	Scans	Lookups
1	5801971…	1	Table_5801	Index_1	CLUSTERED	0	1	920095	3003390844	442375	35632327
2	5801971…	3	Table_5801	Index_3	NONCLUSTERED	0	1	920095	420014	0	0
3	5801971…	7	Table_5801	Index_7	NONCLUSTERED	0	0	920095	2040257472	7	0
4	5801971…	20	Table_5801	Index_20	NONCLUSTERED	0	0	920095	1805673313	43	0
5	5801971…	22	Table_5801	Index_22	NONCLUSTERED	0	0	920095	1288379097	0	0
6	2107310…	1	Table_2107	Index_1	CLUSTERED	0	1	2701711053	136545557	44714994	104653148
7	2107310…	2	Table_2107	Index_2	NONCLUSTERED	0	0	2701711053	2090315	0	0
8	2107310…	3	Table_2107	Index_3	NONCLUSTERED	0	0	2701711053	10	0	0

Reads	Updates	Last Seek	Last Scan	Last Lookup	Last Update
3039465546	1429545988	2021-08-22 09:55:24.943	2021-08-22 09:55:14.567	2021-08-22 09:55:23.983	2021-08-22 09:55:24.923
420014	673402076	2021-08-22 09:54:39.727	NULL	NULL	2021-08-22 09:55:24.923
2040257479	1344552091	2021-08-22 09:55:24.963	2021-02-10 17:43:00.593	NULL	2021-08-22 09:55:24.923
1805673356	1344552091	2021-08-22 09:55:24.990	2021-08-20 10:56:54.933	NULL	2021-08-22 09:55:24.923
1288379097	1344552091	2021-08-22 09:55:25.027	NULL	NULL	2021-08-22 09:55:24.923
285913699	44717469	2021-08-22 09:55:21.800	2021-08-22 09:55:20.300	2021-08-22 09:55:23.583	2021-08-22 09:55:20.297
2090315	44717461	2021-08-22 09:55:04.730	NULL	NULL	2021-08-22 09:55:20.297
10	44717461	2021-07-22 13:15:04.757	NULL	NULL	2021-08-22 09:55:20.297

Рис. 14.2. Вывод представления `sys.dm_db_index_usage_stats`

Рассмотрим столбцы, которые выводит код.

`database_id`, `object_id` *и* `index_id`

В столбцах `database_id`, `object_id` и `index_id` указаны база данных, таблица и индекс соответственно. Их можно использовать для фильтрации вывода, а также объединять с представлениями `sys.databases`, `sys.tables` и `sys.indexes`.

`user_seek`, `user_scan` *и* `user_lookup`

Столбцы `user_seek`, `user_scan` и `user_lookup` показывают, как часто запросы использовали индекс в операторах `Index Seek`, `Index Scan`, `Key Lookup` и `RID Lookup`. Эффективные индексы на основе B-дерева в системах OLTP должны в первую очередь использовать поиск по индексу.

Однако не забывайте о том, что я говорил в главе 5: операторы поиска по индексу не всегда эффективны и могут просматривать очень большой диапазон строк в зависимости от предиката поиска и данных.

`user_update`

В столбце `user_update` показано, сколько раз индекс был изменен операциями вставки, обновления, удаления или слияния. Эта информация позволяет оценить накладные расходы на обслуживание индекса во время модификации данных.

Важно помнить, что `user_update` показывает, сколько раз выполнялась операция, а не сколько строк она изменила. Например, один вызов `DELETE` всегда увеличивает значение `user_update` на 1 независимо от того, сколько строк было удалено.

`last_user_seek`, `last_user_scan`, `last_user_lookup` *и* `last_user_update`

Столбцы `last_user_*` сообщают время последней соответствующей операции в индексе. Эти данные полезны, если SQL Server давно не перезапускался и метрики использования собирались в течение длительного времени.

`system_seek`, `system_scan`, `system_lookup`, `system_update`, `last_system_seek`, `last_system_scan`, `last_system_lookup` *и* `last_system_update`

Столбцы `system_*` (не показаны в сценарии и на рис. 14.2) содержат статистику использования индекса системными процессами. Сюда относятся обновление статистики, обслуживание индекса и некоторые другие процессы. В большинстве случаев об этих показателях не нужно беспокоиться.

Рассмотрим несколько характерных закономерностей в выходных данных представления и разберемся, какие выводы из них можно сделать.

Неиспользуемые индексы

Я проводил много проверок работоспособности SQL Server, и мне редко встречались системы, в которых не было бы неиспользуемых и ненужных индексов. Эти индексы легко обнаружить благодаря тому, что представление `sys.dm_db_index_usage_stats` не показывает по ним никакой активности чтения (`user_seek`, `user_scan` и `user_lookup`).

На рис. 14.3 показан пример такого вывода. (Я удалил некоторые столбцы из результатов листинга 14.10, чтобы вывод стал лаконичнее.) Как видите, на индексах с `index_id` 43 и 47 не выполняется никаких операций чтения.

	index_id	Table	Index	Filtered	Unique	Rows	Seeks	Scans	Lookups	Reads	Updates
1	1	Table_1888	Index_1	0	1	38128199	6464388153	248501	1597518500	8062155154	2169976596
2	21	Table_1888	Index_21	0	0	38128205	4604192	0	0	4604192	153890418
3	40	Table_1888	Index_40	0	0	38128206	0	11	0	11	186522085
4	41	Table_1888	Index_41	0	0	38128206	2263209	40700	0	2303909	288356234
5	42	Table_1888	Index_42	0	0	38128206	561352904	3	0	561352907	186829746
6	43	Table_1888	Index_43	0	0	38128206	0	0	0	0	186526890
7	44	Table_1888	Index_44	0	0	38128206	1681656033	0	0	1681656033	389900596
8	47	Table_1888	Index_47	0	0	38128206	0	0	0	0	1248665902
9	53	Table_1888	Index_53	0	0	38128206	11859	3	0	11862	428364779
10	64	Table_1888	Index_64	1	1	298499	23565855	6193576	0	29759431	1922246283

Рис. 14.3. Неиспользуемые индексы

Как правило, с такими индексами проще всего разбираться. Их можно отключить и/или удалить без особого риска, однако перед этим не забудьте проверить несколько вещей.

Во-первых, как я уже упоминал, убедитесь, что индексы не используются ни на одном узле группы доступности. Очень часто индексы, не используемые на первичном узле, активно задействованы для создания отчетов на доступных для чтения вторичных узлах.

Во-вторых, убедитесь, что статистика использования репрезентативна и что вы не упускаете из виду редкие процессы, которым нужен этот индекс. Тут все не так просто. Необходимо проанализировать преимущества и недостатки сохранения индекса. В некоторых случаях можно попробовать удалить индекс и таким образом устранить его накладные расходы, но тогда редкие процессы будут выполняться неэффективно. Можно также отключить индекс и включать его по расписанию, когда необходимо.

Наконец, уникальные индексы заслуживают особого внимания. Они могут поддерживать уникальные ограничения, и если их удалить, качество данных может испортиться.

Тем не менее во многих случаях неиспользуемые индексы — это «легкая добыча», и их удаление моментально пойдет системе на пользу.

Индексы с дорогостоящим обслуживанием

Индексы, которые гораздо чаще обновляются, чем считываются, я считаю более сложным случаем неиспользуемых индексов. Их удаление может принести

пользу, но сперва стоит проанализировать, как они применяются, и оценить негативные последствия того, что редкие запросы будут выполняться без них.

Я не могу указать формальный порог, после которого следует анализировать индексы. Однако есть несколько критериев, которыми я пользуюсь. Прежде всего я проверяю индексы с небольшим количеством чтений, особенно с небольшим количеством поисков (например, индекс с `index_id=40` на рис. 14.3). Затем я смотрю на индексы, у которых количество обновлений значительно (на порядок или несколько порядков) превышает количество чтений (`index_id=53` на рис. 14.3).

С помощью кода из листинга 14.11 можно просмотреть некоторые запросы, использующие индекс. Этот код анализирует данные кэша планов и может пропустить запросы, план выполнения которых не кэширован. Если у вас включено хранилище запросов, код можно настроить для работы с представлением `sys.query_store_plan` и другими представлениями каталога[1] хранилища запросов.

Сразу оговорюсь, что этот код еще и медленный. Вы можете попробовать сбросить содержимое кэша планов в таблицу служебной базы данных и провести анализ в непромышленной среде.

Листинг 14.11. Обнаружение запросов, использующих индекс

```
DECLARE
    @IndexName SYSNAME = QUOTENAME('<INDEX NAME>');

;WITH XMLNAMESPACES
(DEFAULT 'http://schemas.microsoft.com/sqlserver/2004/07/showplan')
,CachedData
AS
(
    SELECT DISTINCT
        obj.value('@Database','SYSNAME') AS [Database]
        ,obj.value('@Schema','SYSNAME') + '.' + obj.value('@Table','SYSNAME')
            AS [Table]
        ,obj.value('@Index','SYSNAME') AS [Index]
        ,obj.value('@IndexKind','VARCHAR(64)') AS [Type]
        ,stmt.value('@StatementText', 'NVARCHAR(MAX)') AS [Statement]
        ,CONVERT(NVARCHAR(MAX),qp.query_plan) AS query_plan
        ,cp.plan_handle
    FROM
        sys.dm_exec_cached_plans cp WITH (NOLOCK)
            CROSS APPLY sys.dm_exec_query_plan(plan_handle) qp
            CROSS APPLY query_plan.nodes
            ('/ShowPlanXML/BatchSequence/Batch/Statements/StmtSimple')
                        batch(stmt)
            CROSS APPLY stmt.nodes
            ('.//IndexScan/Object[@Index=sql:variable("@IndexName")]') idx(obj)
)
```

[1] https://oreil.ly/ub7Ej

```
SELECT
    cd.[Database]
    ,cd.[Table]
    ,cd.[Index]
    ,cd.[Type]
    ,cd.[Statement]
    ,CONVERT(XML,cd.query_plan) AS query_plan
    ,qs.execution_count
    ,(qs.total_logical_reads + qs.total_logical_writes) / qs.execution_count
        AS [Avg IO]
    ,qs.total_logical_reads
    ,qs.total_logical_writes
    ,qs.total_worker_time
    ,qs.total_worker_time / qs.execution_count / 1000 AS [Avg Worker Time (ms)]
    ,qs.total_rows
    ,qs.creation_time
    ,qs.last_execution_time
FROM
    CachedData cd
        OUTER APPLY
        (
            SELECT
                SUM(qs.execution_count) AS execution_count
                ,SUM(qs.total_logical_reads) AS total_logical_reads
                ,SUM(qs.total_logical_writes) AS total_logical_writes
                ,SUM(qs.total_worker_time) AS total_worker_time
                ,SUM(qs.total_rows) AS total_rows
                ,MIN(qs.creation_time) AS creation_time
                ,MAX(qs.last_execution_time) AS last_execution_time
            FROM sys.dm_exec_query_stats qs WITH (NOLOCK)
            WHERE qs.plan_handle = cd.plan_handle
        ) qs
OPTION (RECOMPILE, MAXDOP 1);
```

Дальнейшие действия будут зависеть от количества запросов, которые используют индекс. Иногда можно провести рефакторинг запросов, переключив их на другой индекс. В других случаях можно удалить индекс, а можно оставить его в базе данных.

Неэффективное чтение

Еще один объект анализа в средах OLTP — индексы с большим количеством просмотров, особенно когда оно значительно выше, чем количество операций поиска. На рис. 14.4 показан такой пример.

Можно использовать подход, аналогичный тому, что рассматривался в предыдущем разделе, и найти запросы, которые используют эти индексы. Я обычно начинаю с индексов, в которых операций поиска очень мало или вообще нет. Запросы, которые просматривают эти индексы, обычно нужно оптимизировать, отчего неэффективное использование индексов может прекратиться.

	index_id	Table	Index	Filtered	Unique	Rows	Seeks	Scans	Lookups	Reads	Updates
1	1	Table_2887	Index_1	0	1	76707068	2516312067	157348	4961245333	7577714748	1398991251
2	3	Table_2887	Index_3	1	0	573442	231677400	0	0	231677400	1387934116
3	4	Table_2887	Index_4	0	0	76707068	21	59382702	0	59382723	854126441
4	5	Table_2887	Index_5	0	0	76707067	3393509	0	0	3393509	824102200
5	6	Table_2887	Index_6	0	0	76707068	1129131537	1	0	1129131538	858627246
6	8	Table_2887	Index_8	0	0	76707068	175291797	3	0	175291800	817555464
7	9	Table_2887	Index_9	0	0	76707068	0	3124770	0	3124770	857147269
8	35	Table_2887	Index_35	0	0	76707068	568834597	0	0	568834597	719052166

Рис. 14.4. Индексы с неэффективным чтением

Также обратите внимание, является ли индекс отфильтрованным. Просмотр отфильтрованных индексов может быть совершенно нормальным явлением, особенно если индекс небольшой.

Неэффективные кластеризованные индексы и кучи

В таблицах с кластеризованными индексами (`index_id=1`) высокое значение `user_lookup` указывает на чрезмерное количество операций поиска по ключу (Key Lookup). Обычно это говорит о том, что либо кластеризованные индексы неэффективны, либо есть некластеризованные индексы, которые не охватывают часто выполняемые запросы.

На рис. 14.5 показан пример. С кластеризованным индексом связано много уточняющих операций (lookup), но не связано ни одной операции поиска (seek). Эта комбинация очень распространена, когда у таблицы есть синтетический `CLUSTERED PRIMARY KEY`, определенный в столбце `IDENTITY`, а запросы используют разные столбцы и индексы для доступа к данным.

	index_id	Table	Index	Filtered	Unique	Rows	Seeks	Scans	Lookups	Reads	Updates
1	1	Table_2067	Index_0	0	0	8362494	0	1	364167805	364167806	285971064
2	2	Table_2067	Index_2	0	1	8362494	364167805	478	0	364168283	195902738
3	3	Table_2067	Index_2	0	1	8362494	27121	315	0	27436	195902738
4	4	Table_2067	Index_2	0	1	8362494	165623	0	0	165623	195902738

Рис. 14.5. Неэффективный кластеризованный индекс

Некоторые из подобных проблем очень легко анализировать и устранять. Например, в ситуации на рис. 14.5 индекс с `index_id=2` явно не охватывает часто выполняемые запросы и порождает операции поиска по ключу. Можно попробовать сделать его кластеризованным индексом, особенно если он не очень широкий. Кроме того, можно включить в индекс дополнительные столбцы, чтобы он охватывал нужные запросы.

К сожалению, не все случаи просты и очевидны. Например, если вы посмотрите на статистику, показанную на рис. 14.4, то увидите, что кластеризованный индекс

используется для поиска как по индексу, так и по ключу. При этом неочевидно, какие некластеризованные индексы могут отвечать за поиск по ключу. В этом случае, вероятно, будет лучше сохранить существующий кластеризованный индекс и проанализировать, к каким некластеризованным индексам обращаются запросы — чтобы, возможно, сделать эти индексы покрывающими.

Наконец, большое количество операций поиска по ключу в таблицах-кучах (`index_id=0`) указывает на чрезмерно интенсивный поиск по RID (RID Lookup). Самый распространенный способ решить эту проблему — превратить один из часто используемых некластеризованных индексов в кластеризованный.

Итоги

Представление `sys.dm_db_index_usage_stats` — отличный инструмент для обнаружения и удаления неэффективных индексов. Однако, как я уже советовал много раз, убедитесь, что вы работаете с данными, которые собраны со всех серверов в группе доступности и охватывают репрезентативный период времени.

Также очень полезно включить в анализ данные представления `sys.dm_db_operational_index_stats`. Рассмотрим его подробнее.

Представление sys.dm_db_index_operational_stats

Представление `sys.dm_db_index_operational_stats` содержит низкоуровневую статистику методов доступа к индексу, блокировок, ввода/вывода и некоторых других областей. Эти данные очень помогают устранять неполадки с производительностью индекса и выявлять узкие места, связанные с обычными и кратковременными блокировками.

В листинге 14.12 показан код, в котором используется это представление. Я привожу его просто для демонстрации, потому что на самом деле лучше объединять эти данные с `sys.dm_db_index_usage_stats` и другими представлениями, о которых я вскоре расскажу.

Листинг 14.12. Представление `sys.dm_db_index_operational_stats`

```
SELECT
    t.object_id
    ,i.index_id
    ,s.name + '.' + t.name AS [Table]
    ,i.name AS [Index]
    ,i.type_desc
    ,i.has_filter AS [Filtered]
    ,i.is_unique AS [Unique]
    ,p.rows AS [Rows]
    ,ous.*
FROM
    sys.tables t WITH (NOLOCK)
```

```
        JOIN sys.indexes i WITH (NOLOCK) ON
            t.object_id = i.object_id
        JOIN sys.schemas s WITH (NOLOCK) ON
            t.schema_id = s.schema_id
        CROSS APPLY
        (
            SELECT SUM(p.rows) AS [rows]
            FROM sys.partitions p WITH (NOLOCK)
            WHERE
                i.object_id = p.object_id AND
                i.index_id = p.index_id
        ) p
        OUTER APPLY sys.dm_db_index_operational_stats
            (DB_ID(),i.object_id,i.index_id,NULL) ous
WHERE
    i.is_disabled = 0 AND
    i.is_hypothetical = 0 AND
    t.is_memory_optimized = 0 AND
    t.is_ms_shipped = 0
ORDER BY
    s.name, t.name, i.index_id
OPTION (RECOMPILE, MAXDOP 1);
```

На рис. 14.6 показан вывод кода. Имена столбцов позволяют понять, какую информацию дает это представление.

Мы не будем обсуждать все эти столбцы. Их подробное описание можно найти в документации Microsoft[1].

Однако мы рассмотрим несколько категорий этих столбцов и обсудим, как их использовать.

Статистика модификации данных

Столбцы `leaf_insert_count`, `leaf_update_count`, `leaf_delete_count`, `leaf_ghost_count` и `leaf_allocation_count` содержат информацию об изменениях данных в индексе. В отличие от представления `sys.dm_db_index_usage_stats`, где подсчитывается количество операций, `sys.dm_db_index_operational_stats` выводит количество затронутых строк. Можно сопоставить информацию из обоих представлений, чтобы лучше оценить накладные расходы на обслуживание индекса.

Кроме того, представление `sys.dm_db_index_operational_stats` содержит в столбцах `nonleaf_*` вышеперечисленные метрики для корневого и промежуточных уровней индексов на основе B-дерева. Эти данные можно использовать, чтобы устранять неполадки кратковременной блокировки `ACCESS_METHODS_HOBT_VIRTUAL_ROOT`, вызванные разбиением корневых страниц и состязанием.

[1] https://oreil.ly/Ze2pc

	object_id	index_id	Table	Index	type_desc	Filtered	Unique	Rows	database_id	object_id	index_id	partition_number
1	1946490013	1	Table_0013	Index_1	CLUSTERED	0	1	0	11	1946490013	1	1
2	274100017	1	Table_0017	Index_1	CLUSTERED	0	1	134	11	274100017	1	1
3	1397580017	1	Table_0017	Index_1	CLUSTERED	0	1	0	NULL	NULL	NULL	NULL

hobt_id	leaf_insert_count	leaf_delete_count	leaf_update_count	leaf_ghost_count	nonleaf_insert_count	nonleaf_delete_count
72057597602774528	0	0	0	0	0	0
72057597681336320	61	0	234	0	3	0
NULL	NULL	NULL	NULL	NULL	NULL	NULL

nonleaf_update_count	leaf_allocation_count	nonleaf_allocation_count	leaf_page_merge_count	nonleaf_page_merge_count	range_scan_count
0	0	0	0	0	4214002
0	3	1	0	0	20302445
NULL	NULL	NULL	NULL	NULL	NULL

singleton_lookup_count	forwarded_fetch_count	lob_fetch_in_pages	lob_fetch_in_bytes	lob_orphan_create_count	lob_orphan_insert_count
0	0	0	0	0	0
3531733818	0	0	0	0	0
NULL	NULL	NULL	NULL	NULL	NULL

row_overflow_fetch_in_pages	row_overflow_fetch_in_bytes	column_value_push_off_row_count	column_value_pull_in_row_count	row_lock_count
0	0	0	0	0
0	0	0	0	594
NULL	NULL	NULL	NULL	NULL

row_lock_wait_count	row_lock_wait_in_ms	page_lock_count	page_lock_wait_count	page_lock_wait_in_ms	index_lock_promotion_attempt_count
0	0	0	0	0	0
0	0	377	0	0	4045
NULL	NULL	NULL	NULL	NULL	NULL

index_lock_promotion_count	page_latch_wait_count	page_latch_wait_in_ms	page_io_latch_wait_count	page_io_latch_wait_in_ms
0	0	0	1	1
0	1	27	1	1
NULL	NULL	NULL	NULL	NULL

tree_page_latch_wait_count	tree_page_latch_wait_in_ms	tree_page_io_latch_wait_count	tree_page_io_latch_wait_in_ms
0	0	0	0
0	0	0	0
NULL	NULL	NULL	NULL

page_compression_attempt_count	page_compression_success_count
0	0
0	0
NULL	NULL

Рис. 14.6. Вывод представления `sys.dm_db_index_operational_stats`

Статистика доступа к данным

Столбцы `singleton_lookup` и `range_scan_count` содержат данные метода доступа. В первом столбце подсчитываются операции поиска и уточнения, которые возвращают отдельные строки. Во втором столбце подсчитываются операции поиска по индексу, которые просматривают диапазон строк вместе с просмотрами индекса. Эти данные можно использовать, чтобы оценить эффективность поиска по индексу на основе значения `singleton_lookup`. Однако невозможно понять, насколько велики диапазоны просмотра, опираясь только на значение `range_scan_count`.

Столбец `forwarded_fetch_count` содержит количество чтений указателей переадресации в таблицах-кучах. Кучи с высоким значением этого столбца неэффективны, и их нужно перестраивать. Этот столбец можно использовать вместе со столбцом `forwarded_record_count` представления `sys.dm_db_index_physical_stats`, которое мы рассматривали ранее в этой главе.

В столбцах lob_fetch_in_pages, lob_fetch_in_bytes, row_overflow_fetch_in_pages и row_overflow_fetch_in_bytes содержится статистика доступа к столбцам вне строки. Запросы к этим таблицам могут выбирать ненужные столбцы — возможно, с помощью антишаблона SELECT *.

Информация о блокировках

Столбцы row_lock_count, row_lock_wait_count, row_lock_wait_in_ms, page_lock_count, page_lock_wait_count и page_lock_wait_in_ms содержат статистику блокировок на уровне строк и страниц. С помощью этой информации можно обнаруживать индексы и таблицы, которые больше всего страдают от проблем с блокировками.

Столбцы index_lock_promotion_attempt_count и index_lock_promotion_count содержат количество попыток укрупнения блокировок и количество их успешных укрупнений на индексе. Последний столбец полезен, когда наблюдается высокий процент ожиданий интентных блокировок (LCK_M_I*), которые часто вызываются укрупнением блокировок.

Обычно я не использую представление sys.dm_db_index_operational_stats в качестве основного инструмента для устранения неполадок с блокировкой и блокированием. Однако оно полезно для перекрестной проверки данных, собранных из других источников, о которых шла речь в главе 8.

Информация о кратковременных блокировках

Столбцы page_latch_wait_count, page_latch_wait_in_ms, tree_page_latch_wait_count и tree_page_latch_wait_in_ms содержат статистику кратковременных блокировок страниц для индекса. Первые два столбца отображают данные для индексных страниц листового уровня, последние два — для промежуточных и корневых страниц.

Эти метрики чрезвычайно полезны, когда наблюдается высокий процент ожиданий PAGELATCH, сгенерированных в базах данных пользователей. Индексы с наибольшим количеством ожиданий кратковременной блокировки страницы, вероятно, укажут на «горячие точки» и узкие места.

Информация о вводе/выводе

Подобно предыдущим категориям, столбцы page_io_latch_wait_count, page_io_latch_wait_in_ms, tree_page_io_latch_wait_count и tree_page_io_latch_wait_in_ms указывают индексы, на которые приходится больше всего ожиданий PAGEIOLATCH.

Как и при устранении неполадок с блокировками, не стоит использовать представление sys.dm_db_index_operational_stats в качестве основного источника данных для устранения неполадок производительности ввода/вывода. Тем не менее оно помогает анализировать индексы с наибольшим количеством ожида-

ний PAGEIOLATCH. Их можно удалить, если они не используются, а можно сжать, чтобы немедленно облегчить нагрузку на ввод/вывод.

Итоги

Представление sys.dm_db_index_operational_stats дает *массу* полезной информации для устранения неполадок. Возможность изучать низкоуровневую статистику по каждому индексу позволяет взглянуть на систему под новым углом, чтобы проверить свои догадки и подтвердить предполагаемые первопричины проблем. Однако не используйте это представление в качестве основного источника выводов.

Представьте, что вы устраняете неполадки производительности дисковой подсистемы. Вы можете обнаружить и удалить некластеризованные индексы с самыми высокими значениями page_io_latch_wait_in_ms, но тогда неоптимизированные запросы просто начнут сканировать другие индексы. Более того, это может привести к еще более интенсивному вводу/выводу, если SQL Server начнет просматривать бóльшие кластеризованные индексы.

Правильнее было бы убедиться, что именно неоптимизированные запросы создают узкие места ввода/вывода, а затем обнаружить и оптимизировать запросы, которые особенно активно эксплуатируют ввод/вывод. При оптимизации запросов данные sys.dm_db_index_operational_stats могут даже не понадобиться, но в некоторых случаях они помогают выявить операторы с интенсивным вводом/выводом в планах выполнения.

Есть одно исключение: кратковременные блокировки страниц. Представление sys.dm_db_index_operational_stats может быть основным инструментом для выявления горячих точек в индексах. Очевидно, понадобится также проанализировать схему и рабочую нагрузку, чтобы подтвердить предположение. В любом случае это представление обычно указывает верное направление и ускоряет устранение неполадок.

Как я уже неоднократно говорил, следует смотреть на систему целостно и использовать все доступные вам инструменты. Представления использования индексов и операционной статистики — отличные средства, которые обязательно должны быть в вашем арсенале.

Целостное представление: sp_Index_Analysis

Представления каталога и динамические административные представления дают немало полезной информации для анализа. К сожалению, этих представлений довольно много, и приходится изучать большое количество разных источников, чтобы получить полную картину. Это неудобно и замедляет работу.

	server	collected_on	database_id	database	guid	object_id	index_id	table	index
1	dbserver	2022-04-24 13:12:52.847	6	AppDB	1	1390627997	1	dbo.tbl_139062799	idx_1
2	dbserver	2022-04-24 13:12:52.847	6	AppDB	0	1348199853	1	dbo.tbl_134819985	idx_1
3	dbserver	2022-04-24 13:12:52.847	6	AppDB	0	93243387	1	dbo.tbl_93243387	idx_1

type	key_columns	included_columns	filter	max_key_length	rows	is_unique	is_disabled	lock_escalation
CLUSTERED	cntnr_name,sc	NULL	NULL	526	180173162	1	0	TABLE
CLUSTERED	delta_seq_num	NULL	NULL	8	179977854	1	0	DISABLE
CLUSTERED	delta_seq_num	NULL	NULL	8	116448546	1	0	TABLE

max_compression	total_pages	used_pages	data_pages	total_space_mb	used_space_mb	data_space_mb	buffer_pool_space_mb
NONE	1530884	1515056	1494061	11960.031	11836.375	11672.352	17.148
NONE	2422980	2422495	2411026	18929.531	18925.742	18836.141	NULL
NONE	3944236	3943575	3924964	30814.344	30809.180	30663.781	NULL

stats_date	user_seeks	user_scans	user_lookups	user_reads	user_updates	last_user_seek
2022-04-24 03:08:00	212675105	23	0	212675128	171501834	2022-04-24 13:02:00
2022-04-24 07:00:00	31833	8	0	31841	40149	2022-04-24 13:00:00
2022-04-24 07:04:00	31833	0	0	31833	4946	2022-04-24 13:00:00

last_user_scan	last_user_lookup	last_user_update	range_scan_count	singleton_lookup_count	forwarded_fetch_count
2022-03-18 09:58:0	NULL	2022-04-24 13:02	10932050	212675090	0
2022-03-18 15:27:0	NULL	2022-04-24 06:10	0	19877031	0
NULL	NULL	2022-04-24 06:19	0	13433214	0

lob_fetch_in_pages	row_overflow_fetch_in_pages	leaf_insert_count	leaf_update_count	leaf_delete_count	leaf_ghost_count
0	0	130328590	41173243	0	0
0	0	10329936	0	0	9547095
0	0	6671993	0	0	6761221

nonleaf_insert_cou	nonleaf_update_cou	nonleaf_delete_coun	leaf_allocation_count	nonleaf_allocation_count	row_lock_count
2882818	53525310	1804263	12961165	15147	305265513
414946	0	383109	414946	1957	21489764
673193	0	659906	673193	3175	299088

row_lock_wait_cour	row_lock_wait_in_	page_lock_cou	page_lock_wait_cou	page_lock_wait_in	index_lock_promotion_attempt_c
0	0	297383904	40	654	0
0	0	30222224	0	0	0
0	0	1680475	0	0	523

index_lock_promotion_cou	page_latch_wait_cou	page_latch_wait_in_ms	tree_page_latch_wait_cou	tree_page_latch_wait_in_ms
0	3216	4743	2396	4317
0	1	1	1	1
37	5	23	3	18

page_io_latch_wait_count	page_io_latch_wait_in_ms	page_compression_attempt_count	page_compression_success_count
5071739	8374927	0	0
48026	45560	0	0
21728	11558	0	0

Рис. 14.7. Вывод `sp_Index_Analysis`

Я для себя решил эту проблему, написав хранимую процедуру `sp_Index_Analysis`, которая объединяет информацию из различных представлений и возвращает ее в виде одного вывода. Ее код можно загрузить из сопутствующих материалов этой книги или с моего сайта[1].

Хранимая процедура содержит большой объем информации, в том числе:

- определение индекса и его метаданные;
- размер индекса на диске и в буферном пуле;
- статистику использования индекса;
- операционную статистику индекса;
- статистическую информацию.

Пример вывода процедуры показан на рис. 14.7.

Моя хранимая процедура собирает информацию из одной или нескольких баз данных и позволяет сохранить выходные данные в таблице для дальнейшего анализа. Я обычно свожу в одну таблицу данные, собранные со всех узлов группы доступности, прежде чем начинать их исследовать.

Естественно, вы не обязаны пользоваться этой хранимой процедурой. Однако она ускорит устранение неполадок и упростит вашу работу.

Резюме

Представления каталога и динамические административные представления SQL Server — это богатейшие источники информации, которые помогают выявить очаги неэффективности в конструкции базы данных и стратегии индексирования. Имеет смысл анализировать схему базы данных и использование индекса в рамках проверки работоспособности системы.

Чтобы обнаружить дефекты конструкции базы данных, можно использовать основные системные каталоги, такие как `sys.tables`, `sys.indexes`, `sys.index_columns`, `sys.foreign_keys` и др. В числе возможных проблем могут быть неэффективные таблицы-кучи и кластеризованные индексы, неиндексированные ограничения внешних ключей и избыточные индексы.

Всегда проверяйте, не приближаются ли столбцы `IDENTITY` к максимальной емкости типа данных. Если эта емкость исчерпается, это может привести к длительному простою.

[1] https://aboutsqlserver.com/

Представление `sys.dm_db_index_usage_stats` позволяет анализировать и обнаруживать неэффективные индексы в базе данных. Сюда относятся, в частности, неиспользуемые и неоптимальные индексы, индексы с высокими затратами на обслуживание и неэффективные кластеризованные индексы. Еще одно представление — `sys.dm_db_index_operational_stats` — содержит низкоуровневую статистику доступа, блокировок и ввода/вывода. Оно помогает обнаружить «горячие точки», связанные с кратковременными блокировками страницы, и найти индексы, которые порождают другие проблемы с производительностью. Оба представления очищаются при перезапуске SQL Server и при отключении базы данных. Они не отражают использование индексов на вторичных репликах группы доступности. Во время анализа убедитесь, что вы работаете с репрезентативной информацией.

Наконец, хранимая процедура `sp_Index_Analysis` предоставляет целостную информацию об индексах, включая их метаданные, размер на диске и в буферном пуле, а также метрики использования. Процедуру `sp_Index_Analysis` можно загрузить из сопутствующих материалов книги.

Настало время обсудить виртуализацию, а также устранение неполадок SQL Server в виртуализированной среде.

Чек-лист устранения неполадок

- ☐ Выявить потенциальные дефекты схемы базы данных с помощью представлений каталога.
- ☐ Просмотреть текущие значения и оставшуюся емкость столбцов `IDENTITY`.
- ☐ Проанализировать использование индексов с помощью представлений `sys.dm_db_index_usage_stats` и `sys.dm_db_index_operational_stats` и решить возможные проблемы.

ГЛАВА 15

SQL Server в виртуализированных средах

Виртуализация сегодня настолько распространена, что чаще всего экземпляры SQL Server, на которых мы устраняем неполадки, работают в виртуальных средах, а это добавляет определенные сложности.

Я начну эту главу с некоторых соображений о виртуализации баз данных в целом. Далее я расскажу об эффективных методах настройки SQL Server на виртуальных машинах. Наконец, я дам несколько советов по устранению неполадок на виртуальных экземплярах SQL Server.

Не нужно считать эту главу исчерпывающим руководством по виртуализации SQL Server. Ее цель — дать вам достаточно знаний для того, чтобы выявлять неэффективные настройки SQL Server и устранять распространенные проблемы с производительностью в виртуальных средах.

Также я введу вас в контекст, который позволит говорить на одном языке со специалистами, отвечающими за инфраструктуру виртуализации. Наилучшие результаты достигаются, когда разные команды работают вместе и обмениваются опытом. Налаживая и поддерживая виртуальные экземпляры SQL Server, сотрудничайте с командой виртуализации.

Виртуализировать или не виртуализировать — вот в чем вопрос

Вопрос о том, надо ли виртуализировать среды SQL Server, всегда вызывал споры. Пятнадцать лет назад я был ярым противником виртуализации, но с тех пор передумал. В любом случае наше мнение мало что значит: виртуализация дает множество преимуществ, она широко используется, и специалисты по базам данных регулярно имеют дело с виртуализированными экземплярами SQL Server.

Одно из преимуществ виртуализации — в том, что она снижает затраты на инфраструктуру. Можно развернуть несколько виртуальных машин на одном физическом оборудовании и использовать его гораздо экономичнее.

Кроме того, виртуализация значительно упрощает обслуживание системы. Можно увеличивать или уменьшать мощность виртуальных машин с помощью простых изменений конфигурации. Можно без особых усилий модернизировать оборудование, переместив виртуальную машину на новый, более мощный хост. Можно без простоев перенести данные в более быстрое хранилище, просто скопировав виртуальные диски.

Наконец, виртуализация может обеспечить еще один уровень *высокой доступности* (*HA*) за счет перезапуска или даже перемещения виртуальных машин на другие хосты в случае аппаратного сбоя. При разработке стратегии высокой доступности я не рекомендую полагаться исключительно на виртуализацию, но она может работать как дополнительный уровень защиты в сочетании с собственными технологиями высокой доступности SQL Server.

К сожалению, эти преимущества связаны со снижением производительности. Величина накладных расходов бывает разной: они невелики в небольших системах, но растут вместе с размером виртуальных машин. Например, в крупных виртуальных машинах с десятками ядер и терабайтами памяти накладные расходы могут достигать 10 % или даже больше.

Виртуализация также влияет на стоимость лицензирования. В SQL Server Enterprise Edition виртуализация сэкономит вам немало денег, если вы лицензируете SQL Server на уровне хоста и выделяете избыточные ресурсы (о которых я скоро расскажу). В то же время, если вы используете лицензирование по количеству ядер и/или хотите избежать избыточного выделения ресурсов, то накладные расходы на производительность виртуализации заставят вас добавить больше ЦП виртуальной машине, для чего понадобится покупать дополнительные лицензии.

В конце концов, вам нужно исходить из того, на какую стоимость лицензии вы ориентируетесь, и решить, перевешивают ли преимущества виртуализации связанные с ней потери производительности. Тут все зависит от конкретной ситуации, и для правильного решения вам необходимо опираться на свой опыт работы с базами данных.

На мой взгляд, виртуализация — это обычно правильное решение, за исключением редких особых случаев и критических систем, где нужно получить максимальную отдачу от оборудования.

Есть одна задача, для которой я считаю виртуализацию особенно полезной: консолидация баз данных. Гораздо проще поддерживать несколько виртуальных машин SQL Server, консолидированных на общем хосте, чем несколько баз данных, работающих на одном экземпляре SQL Server.

В этом сценарии виртуализация изолирует системы друг от друга. Можно использовать разные версии SQL Server и разные ОС, применять отдельные графики обслуживания, поддерживать требования безопасности и соответствия, специфичные для той или иной системы, а также уменьшить влияние, которое оказывают на производительность функции уровня экземпляра, такие как прозрачное шифрование данных (которое также шифрует базу данных `tempdb`) или аудит.

Также учитывайте вашу модель лицензирования SQL Server. Standard Edition требует лицензировать *логические ядра* в виртуальных машинах, что может оказаться дороже, чем один физический экземпляр SQL Server. Вместе с тем Enterprise Edition позволяет лицензировать *физические ядра* на хосте независимо от того, сколько логических ядер выделено для виртуальных машин баз данных.

Давайте обсудим оснащение виртуальных машин и устранение соответствующих неполадок, начиная с настройки виртуальных машин SQL Server.

Настройка SQL Server в виртуализированных средах

Накладные расходы на виртуализацию довольно малы, но только если виртуальные машины правильно оснащены и настроены. Из-за неграмотно реализованной виртуализации производительность серверов баз данных может серьезно снизиться.

Налаживая виртуализацию, необходимо обратить внимание на несколько аспектов, среди которых — планирование мощности, конфигурация ЦП, память, хранилище и сеть. Рассмотрим их по порядку.

Планирование мощности

Одна из основных целей виртуализации — снизить затраты на оборудование. Специалисты по инфраструктуре и администраторы виртуализации стараются размещать на хостах как можно больше виртуальных машин, используя физическое оборудование по максимуму, но эта стратегия не всегда оптимальна для развертывания баз данных. Залог успешной виртуализации SQL Server — правильное планирование мощности на уровне как хоста, так и виртуальной машины.

Все гостевые виртуальные машины совместно используют ресурсы хоста и конкурируют за них. Можно выделять и резервировать ресурсы (главным образом ЦП и ОЗУ) для отдельных виртуальных машин, но администраторы виртуализации предпочитают этого не делать и часто *перевыделяют (overcommit)*

ресурсы. Например, на хосте с 32 физическими ядрами и 512 Гбайт ОЗУ может работать два десятка виртуальных машин, на которые в совокупности выделено вдвое больше логических ядер и ОЗУ.

Перевыделение ресурсов снижает потребность в физическом оборудовании и экономит деньги. Это может быть приемлемо для многих некритических баз данных, работающих с небольшой нагрузкой, но в то же время перевыделение может серьезно снизить производительность критически важных баз данных, потому что виртуальные машины конкурируют за ресурсы. Могут возникнуть ненужные задержки, пока виртуальные машины ожидают доступности ЦП, а это приводит к нехватке памяти в виртуальных машинах и к другим узким местам. Для критически важных виртуальных машин лучше резервировать ресурсы.

Однако у резервирования ресурсов тоже есть негативные последствия. Например, в некоторых ситуациях оно ограничивает возможность перемещать виртуальную машину на другой хост, не выключая ее. Обсудите это со своими специалистами по инфраструктуре: возможно, они предложат другие варианты того, как расставить приоритеты рабочей нагрузки и использовать ресурсы на критически важных виртуальных машинах.

Беседуя со специалистами по инфраструктуре, будьте осторожны с терминологией. Например, термин *перевыделение* (*overcommitment*) означает разные вещи для специалистов по базам данных и по инфраструктуре. Когда я впервые обсуждал эту тему с администраторами виртуализации, я имел в виду, что количество выделенных логических ресурсов превышает доступные физические ресурсы хоста. Но оказалось, что администратор виртуализации не считал ресурсы «перевыделенными» до тех пор, пока они не преодолели соотношение 3:1.

Базы данных нужно анализировать с точки зрения их рабочей нагрузки, соглашения об уровне обслуживания (SLA) и критичности для бизнеса. Обычно я выделяю три уровня:

Уровень 1

Критически важные базы данных, работающие под большой нагрузкой. Виртуальные машины для этих баз не должны работать на хостах с перевыделением, или у них должно быть включено резервирование ресурсов. На этом уровне также нужно обратить внимание на конфигурацию процессора, о которой я вскоре расскажу.

Уровень 2

Промышленные базы данных, работающие со средней нагрузкой. Они часто нормально функционируют при некотором уровне перевыделения на хосте. Они также могут потреблять ресурсы в приоритетном порядке по сравнению с менее важными виртуальными машинами на том же хосте.

Уровень 3

Базы данных, которые не являются критически важными, работают при небольшой нагрузке и не имеют соглашений об уровне производительности. На хостах с такими базами ресурсы обычно перевыделены.

Если вы используете подобную классификацию баз данных, кооперируйтесь с администраторами виртуализации, чтобы разработать правильную топологию виртуальных машин и их конфигурацию. Очевидно, что критически важные базы данных уровня 1 требуют максимального внимания и самого тщательного планирования.

В общем случае следует избегать перевыделения ресурсов для гостевых машин в виртуальных средах. Лучше и проще выделить виртуальным машинам столько ресурсов, сколько нужно для текущей нагрузки, а затем добавлять дополнительные ресурсы по мере роста нагрузки. Убедитесь, что у вас настроен эффективный мониторинг, и заблаговременно определяйте, когда пора наращивать мощность.

К счастью, цена ошибок и неверных конфигураций в виртуальных средах намного ниже, чем при использовании физического оборудования. Не составляет труда добавить дополнительных ресурсов к «обделенным» виртуальным машинам и перераспределить виртуальные машины между хостами по мере необходимости. Имейте это в виду при планировании мощности.

Конфигурация ЦП

Я начну с контринтуитивного заявления: в общих средах добавление процессоров к виртуальным машинам может *снизить* их производительность, а не *улучшить* ее. Во многих случаях виртуальная машина с меньшим количеством ядер будет работать быстрее, чем с бóльшим, — однако при условии, что у выделенных ядер хватает пропускной способности, чтобы обрабатывать нагрузку.

Все дело в том, как в гипервизорах устроено планирование. В этом отношении алгоритмы работы Hyper-V и VMware немного различаются.

Накладные расходы на планирование и особенности реализации приводят к простому правилу: *в общих средах, особенно с перевыделением, выделяйте ровно столько ЦП, сколько нужно для имеющейся нагрузки*. После этого следите за производительностью и добавляйте ЦП по мере роста нагрузки. Это полезно и в VMware, и в Hyper-V, хотя VMware более чувствительна к избыточному выделению ЦП.

Тем не менее избыточное оснащение ресурсами может не вызывать проблем для критически важных виртуальных машин, если они используют резервирование ресурсов или запускаются на хостах без перевыделения. Нужно только проверить, что топология со временем не меняется. Мне встречались администраторы виртуализации, которые добавляли новые виртуальные машины к критическим хостам, не сообщая об этом специалистам по базам данных.

ПАРА СЛОВ О ПЛАНИРОВАНИИ

Внедрить эффективное планирование для многопроцессорных виртуальных машин — это всегда трудная задача. Нужно синхронизировать потоки, работающие на разных ЦП, и невозможно выполнять рабочую нагрузку виртуальной машины только на подмножестве ЦП.

В первой версии гипервизора VMware использовалась концепция *бригадного планирования (gang scheduling)*. В этой модели гипервизор не позволяет виртуальной машине выполняться, если у него не хватает физических ЦП, чтобы назначить их всем ЦП в виртуальной машине. Например, машина, на которую выделено четыре ЦП, будет оставаться приостановленной до тех пор, пока на хосте для нее не найдется четырех свободных процессоров.

Бригадное планирование довольно хорошо работало с небольшими виртуальными машинами, но плохо масштабировалось на крупные многопроцессорные виртуальные системы, поэтому в VMware появилась другая модель планирования. При *ослабленном совместном планировании (relaxed co-scheduling)* гипервизор позволяет процессорам в крупных виртуальных машинах работать с большей степенью независимости. При этом гипервизор отслеживает работу процессоров и лимитирует их при необходимости.

Но и эта модель планирования неидеальна. Производительность по-прежнему ограничена процессором с самой низкой частотой и пропускной способностью, и все еще бывают случаи, когда гипервизору приходится планировать одновременную работу всех ЦП, как при бригадном планировании.

В то же время в Hyper-V используется другая модель, которая опирается на возможности гостевой ОС. Современные версии Windows и Linux понимают, когда они работают из-под виртуальной машины, и ядро в гостевой ОС запрашивает у гипервизора процессорное время. Эта модель оптимизирована так, чтобы уменьшить (но не полностью устранить) необходимость синхронизировать планирование между процессорами. Она более эффективна в случае больших виртуальных машин и перевыделения ЦП.

В конфигурации ЦП в виртуализированных средах участвуют несколько компонентов. Вот важные компоненты хоста:

Физический ЦП (`pCPU`*)*

Это физический процессорный чип, установленный на сервере. Его иногда называют *физическим сокетом* (`pSocket`).

Физическое ядро (`pCore`*)*

Это независимое вычислительное ядро в физическом ЦП.

Логическое ядро (`lCore`*)*

Это логическая вычислительная единица на физическом ядре, которая также называется *логическим процессором*. При гиперпоточности на каждое физи-

ческое ядро приходятся два логических ядра. Общее количество логических процессоров на хосте можно рассчитать как (количество `pCPU`) * (количество `pCore` на один `pCPU`) * (количество `lCore` на один `pCore`).

Например, у хоста с двумя процессорами Intel Xeon Gold 6346 с включенной гиперпоточностью будет 2 `pCPU` * 16 `pCore` * (2 `lCore` на один `pCore`) = = 64 логических ядра.

А это важные компоненты виртуальной машины:

Виртуальный сокет (`vSocket`) и виртуальное ядро (`vCore`)

Виртуальный сокет — это виртуализированная микросхема ЦП, которая может быть сконфигурирована с одним или несколькими виртуальными ядрами. В виртуальных машинах виртуальные сокеты обрабатываются как физические сокеты и, в свою очередь, как отдельные узлы NUMA. (Подробнее об этом чуть позже.)

Виртуальный ЦП (`vCPU`)

Это виртуальный процессор, назначенный виртуальной машине. Как вы наверняка догадались, общее количество виртуальных ЦП в виртуальной машине можно рассчитать как (общее количество `vSocket`) * (количество `vCore` на один `vSocket`).

Есть несколько вопросов, которые следует рассмотреть при настройке виртуальных машин для SQL Server. Первый вопрос — включать ли гиперпоточность. На него не так просто ответить, когда вы имеете дело с виртуализацией.

Гиперпоточность может повысить пропускную способность хоста, но также может снизить пропускную способность отдельных виртуальных машин. Теоретически хост учитывает гиперпоточность во время планирования, назначая виртуальные ЦП физическим ядрам. Но на практике все может зависеть от загруженности хоста.

Как обычно, стоит протестировать производительность с включенной гиперпоточностью и без нее. К сожалению, эффект в виртуальных средах трудно измерить, потому что нагрузка на хост быстро меняется. Гиперпоточность может неплохо подходить для некритических рабочих нагрузок, но на виртуальных машинах SQL Server уровня 1 безопаснее оставить ее отключенной.

Второй важный вопрос — конфигурация NUMA. Windows и SQL Server Enterprise Edition поддерживают NUMA. Согласование конфигурации виртуальных NUMA на гостевых виртуальных машинах с конфигурацией физического NUMA (`pNuma`) может повысить производительность.

Очевидно, лучше протестировать и отрегулировать конфигурацию в соответствии с вашей реальной рабочей нагрузкой. Мое эмпирическое правило для

начальной конфигурации заключается в том, что с конфигурацией NUMA нужно согласовывать от 8 до 12 виртуальных ядер на каждый виртуальный сокет. Занимаясь этим согласованием, помните, что Enterprise Edition по умолчанию использует программный NUMA, если на узле NUMA больше восьми планировщиков.

Мой подход может измениться, если на хосте нет перевыделения, а требования к ЦП и ОЗУ виртуальной машины укладываются в один узел NUMA. В этом случае я могу держать виртуальную машину в пределах одного физического узла NUMA.

Рассмотрим несколько примеров. Предположим, что у вас есть сервер с двумя 16-ядерными процессорами и 256 Гбайт ОЗУ на каждый узел NUMA. В табл. 15.1 показаны возможные конфигурации виртуальных машин в зависимости от их требований.

Таблица 15.1. Возможные конфигурации виртуальных машин

Конфигурация виртуальной машины	Количество vCore и vSocket	Примечание
8 ЦП 128 Гбайт ОЗУ	1 vSocket 8 vCores на vSocket	Все укладывается в один узел pNuma
12 ЦП 192 Гбайт ОЗУ	1 vSocket, 8 vCores на vSocket или 2 vSocket и 6 vCores на vSocket	Протестируйте и выберите оптимальную конфигурацию
16 ЦП 256 Гбайт ОЗУ	2 vSocket, 8 vCores на vSocket или 1 vSocket и 16 vCores на vSocket	Протестируйте и выберите оптимальную конфигурацию. Первый вариант дает хосту больше гибкости и может обеспечить лучшую производительность при перевыделении
8 ЦП 384 Гбайт ОЗУ	2 vSocket 4 vCores на vSocket	Требует памяти от двух узлов NUMA
20 ЦП 256 Гбайт ОЗУ	2 vSocket 10 vCores на vSocket	Требует ядер от двух узлов NUMA

Убедитесь, что все узлы NUMA в SQL Server правильно согласованы и имеют одинаковое количество процессоров. Как упоминалось в главе 2, SQL Server назначает соединения с узлами NUMA в циклическом режиме. Неравномерное распределение ЦП между узлами приведет к проблемам с планированием, потому что некоторые планировщики в этой конфигурации будут выполнять

больше работы и станут более нагруженными, чем другие. Наличие этой проблемы можно проверить с помощью кода из листинга 15.1.

Листинг 15.1. Проверка распределения планировщиков в узлах NUMA

```
SELECT
    parent_node_id AS [NUMA Node]
    ,COUNT(*) AS [Schedulers]
    ,SUM(IIF(status = N'VISIBLE ONLINE',1,0))
        AS [Online Schedulers]
    ,SUM(IIF(status = N'VISIBLE OFFLINE',1,0))
        AS [Offline Schedulers]
    ,SUM(current_tasks_count)
        AS [Current Tasks]
    ,SUM(runnable_tasks_count)
        AS [Runnable Tasks]
FROM
    sys.dm_os_schedulers WITH (NOLOCK)
WHERE
    status IN (N'VISIBLE ONLINE',N'VISIBLE OFFLINE')
GROUP BY
    parent_node_id
OPTION (RECOMPILE, MAXDOP 1);
```

Некоторое время назад мне пришлось устранять неполадки с производительностью в виртуальной машине, которая работала на хосте с двумя 18-ядерными процессорами. Машина была снабжена 17 виртуальными сокетами по 2 виртуальных ядра на каждый сокет. В этой конфигурации Windows увидела 3 физических узла NUMA с 16, 16 и 2 виртуальными ЦП соответственно. SQL Server, в свою очередь, использовал программный NUMA и разделил первые 2 узла пополам.

На рис. 15.1 показан вывод листинга 15.1 в этой конфигурации. На пятом узле NUMA оказалось два планировщика, которые были нагружены значительно сильнее остальных.

	Numa Node	Schedulers	Online Schedulers	Offline Schedulers	Current Tasks	Runnable Tasks
1	0	8	8	0	43	10
2	1	8	8	0	38	2
3	2	8	8	0	40	7
4	3	8	8	0	34	1
5	4	2	2	0	77	51

Рис. 15.1. Узел NUMA и планировщики в несбалансированной конфигурации

На рис. 15.2 показан график загрузки процессора на сервере. На графике видно, что эти два ЦП были загружены до предела, что приводило к периодическим проблемам с производительностью и скачкам ожиданий `HADR_SYNC_COMMIT`.

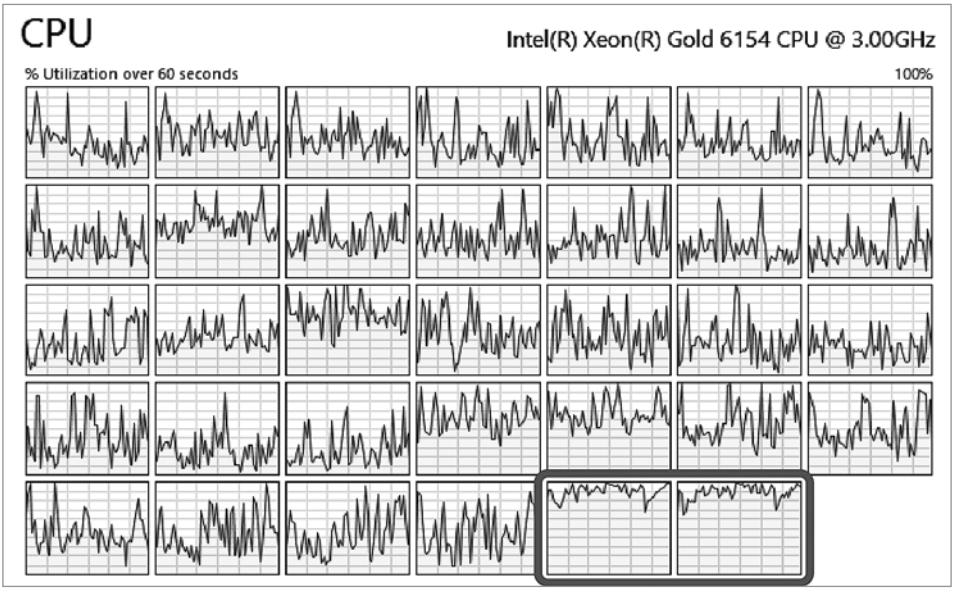


Рис. 15.2. Нагрузка на ЦП в несбалансированной конфигурации

Мы переконфигурировали виртуальную машину: настроили на ней 2 виртуальных сокета по 17 виртуальных ядер на сокет и согласовали ее с конфигурацией физического оборудования. В этом режиме Windows видела два узла NUMA, однако SQL Server разделил ЦП на два 5-ядерных и четыре 6-ядерных процессора. Конфигурация снова получилась несовершенной, но все же более сбалансированной, чем первоначальная.

В конце концов мы облегчили виртуальную машину до 32 виртуальных ЦП, что дало четыре 8-ядерных программных узла NUMA, идеально согласованных с оборудованием. Сперва мы думали сохранить 34 виртуальных ЦП и отключить программный NUMA, но отказались от этого, потому что у виртуальной машины хватало мощности для рабочей нагрузки.

Соотношение виртуальных сокетов и виртуальных ядер на сокет (`vSocket/vCore`) особенно важно в версиях SQL Server, отличных от Enterprise, в которых количество сокетов ограниченно. Неправильные настройки могут помешать им использовать все виртуальные ЦП в виртуальной машине. Эту ситуацию легко обнаружить, наблюдая за загрузкой ЦП на сервере: вы увидите, что одни ЦП заняты, а другие бездействуют. Эту информацию также можно получить из столбца `[OfflineSchedulers]` в выходных данных листинга 15.1. Кроме того, можно изучить столбец `status` в представлении `sys.dm_os_schedulers`. На рис. 15.3 показан вывод этого представления в Standard Edition с несколькими отключенными планировщиками.

	scheduler_address	parent_node_id	scheduler_id	cpu_id	status	is_online
1	0x00000200D9220040	0	0	0	VISIBLE ONLINE	1
2	0x00000200D9240040	0	0	1	VISIBLE ONLINE	1
3	0x00000200D9260040	0	0	2	VISIBLE ONLINE	1
4	0x00000200D9280040	0	0	3	VISIBLE ONLINE	1
5	0x00000200D92A0040	0	0	4	VISIBLE OFFLINE	0
6	0x00000200D92C0040	0	0	5	VISIBLE OFFLINE	0
7	0x00000200D9C20040	0	0	6	VISIBLE OFFLINE	0
8	0x00000200D9C40040	0	0	7	VISIBLE OFFLINE	0
9	0x00000200D9C60040	0	0	8	VISIBLE OFFLINE	0
10	0x00000200D9C80040	0	0	9	VISIBLE OFFLINE	0
11	0x00000200D9CA0040	0	0	10	VISIBLE OFFLINE	0

Рис. 15.3. Статус планировщиков

Проблема с несбалансированными конфигурациями NUMA и планировщиками может возникнуть и у невиртуализированных экземпляров SQL Server, если некоторые планировщики были отключены с помощью маски соответствия или если используется версия Enterprise Edition с лицензиями клиентского доступа, которые ограничивают ее до 20 физических ядер. Но гораздо чаще эта проблема проявляется в виртуальных средах, позволяющих создавать конфигурации виртуальных машин с мощностями, которых нет на физическом оборудовании. Не забудьте проконтролировать это во время проверки работоспособности.

Память

Как и в случае с конфигурацией ЦП, виртуальным машинам (VM) можно выделить больше памяти, чем ее физически есть на хосте, однако это может привести к серьезным проблемам с производительностью серверов баз данных.

Если вы не используете резервирование ресурсов, хост не выделяет память для VM сразу: она выделяется только тогда, когда приложения в VM ее запрашивают. Однако, когда приложения освобождают эту память для ОС, хост об этом не знает, поэтому виртуальные машины продолжают потреблять память. По мере того, как объем выделенной памяти на хосте растет, хост начинает отзывать ее у гостевых ОС с помощью *драйвера накачки (balloon driver)*. Этот драйвер устанавливается на гостевой виртуальной машине и запрашивает память у своей ОС, возвращая ее хосту. Это может привести к нехватке внешней памяти в SQL Server и, возможно, к подкачке внутри ОС.

По умолчанию SQL Server реагирует на нехватку памяти, сокращая ее потребление и освобождая память для операционной системы, но это все равно ведет

к снижению производительности. Ситуация усугубляется, если у вас включена *блокировка страниц в памяти* (*LPIM, lock pages in memory*) или неправильно установлен *минимальный объем памяти сервера*. Эти настройки могут помешать SQL Server высвобождать память, что может привести к нестабильности ОС и аварийным переключениям.

С этим мало что можно поделать, помимо правильного планирования мощности. Не выделяйте лишнюю память для виртуальных машин, если только вы не используете резервирование ресурсов или не можете гарантировать, что на хосте не будет перевыделения. SQL Server лучше управляется с меньшим объемом доступной памяти, чем с регулярной нехваткой памяти из-за накачки.

Будьте осторожны с LPIM и другими функциями, которые могут помешать SQL Server реагировать на нехватку памяти. Не забывайте, что хост может внезапно оказаться перевыделенным, если на него мигрируют другие виртуальные машины, что бывает при балансировке нагрузки виртуальных машин или во время аварии, когда другие хосты в инфраструктуре выходят из строя.

Наконец, не забывайте, что конфигурация памяти может влиять на конфигурацию ЦП в виртуальной машине. Если вашей виртуальной машине требуется память из нескольких узлов `pNuma`, убедитесь, что конфигурация `vSocket/vCore` соответствует этой топологии.

Хранилище

У всех платформ виртуализации есть различные варианты настройки хранилища. Чаще всего виртуальные машины используют *виртуальные диски* (`vmdk` в VMware и `vhdx` в Hyper-V), представленные как отдельные диски внутри гостевых виртуальных машин. Хост размещает виртуальные диски либо непосредственно в физическом хранилище, либо, как VMware vSphere, в другом хранилище данных, оптимизированном для виртуальных дисков (рис. 15.4).

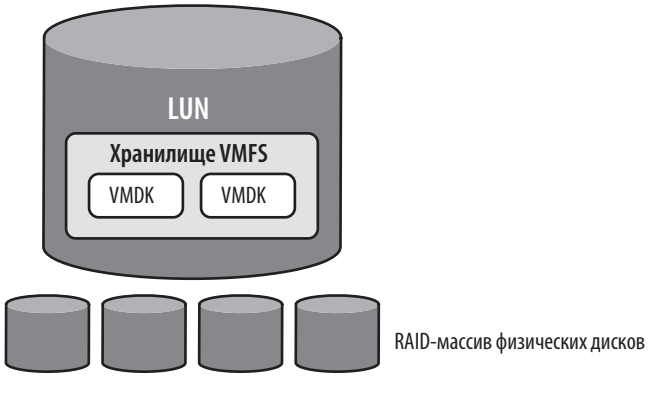


Рис. 15.4. Стек хранилища VMware vSphere

Можно напрямую отобразить физическое хранилище на виртуальную машину, минуя виртуальные диски, но я редко рекомендую этот вариант. Он не дает заметного прироста производительности по сравнению с правильно настроенным виртуальным хранилищем, зато вы теряете гибкость виртуализированного хранилища и возможность перемещать данные между массивами хранилища.

В основном конфигурация хранилища зависит от инфраструктуры системы и настраивается за пределами виртуальной машины SQL Server, но в системах уровня 1 нужно разделять рабочую нагрузку ввода/вывода между различными контроллерами SCSI и хранилищами данных. Это звучит похоже на старинную рекомендацию распределять объекты базы данных по нескольким дискам, но цель в другом: такой прием помогает сбалансировать запросы ввода/вывода и уменьшает насыщение дисковых очередей во время всплесков рабочей нагрузки ввода/вывода. Одним из примеров служит процесс контрольной точки, который может генерировать огромное количество запросов на запись и увеличивать задержку ввода/вывода (см. главу 3).

Держите все связанные хранилища данных в одной группе защиты хранилища, чтобы избежать повреждения данных в случае аварии.

Существует ограничение на количество виртуальных контроллеров SCSI, которые может поддерживать каждая виртуальная машина. Например, в VMware можно использовать только четыре контроллера PVSCSI на каждую виртуальную машину. Обычно я помещаю диск журнала транзакций на один из контроллеров, объединяю диски `tempdb` и ОС на другом, а два оставшихся контроллера использую для файлов данных. Однако иногда я поступаю по-другому в зависимости от количества баз данных на сервере, количества файлов данных в базах данных и распределения рабочей нагрузки ввода/вывода между файлами. Выбирая оптимальную конфигурацию, можно проанализировать счетчики производительности диска и выборочные данные в представлении `sys.dm_io_virtual_file_stats`.

Увеличьте и отрегулируйте глубину очереди в контроллерах SCSI и используйте последние доступные версии драйверов. Убедитесь, что вы применяете оптимальный тип контроллера для конфигурации хранилища. Например, VMware vSphere предоставляет контроллер vNVMe, оптимизированный для локальных флеш-накопителей.

Как и в случае с процессором и памятью, у хоста и хранилища должно хватать пропускной способности для обработки нагрузки ввода/вывода. Не рекомендуется объединять несколько виртуальных машин с интенсивным вводом/выводом на одном хосте или размещать их в одних и тех же хранилищах данных, потому что они могут влиять друг на друга. Имейте в виду, что топология

виртуальной машины может меняться со временем, и не забывайте следить за производительностью.

Для снижения рисков можно использовать решения Quality of Service (QoS), например Virtual Volumes (VVols) в vSphere или Storage QoS в Hyper-V. Они могут смягчить влияние «шумных соседей» и сделать рабочую нагрузку ввода/вывода более предсказуемой.

Большинство проблем с конфигурацией возникает, когда виртуальные машины SQL Server оснащаются на основании общих соображений, без учета поведения SQL Server и характерных шаблонов ввода/вывода. Этих проблем легко избежать, если вы сотрудничаете с администраторами систем виртуализации и хранения данных.

Всегда рекомендуется по возможности проверять производительность ввода/вывода на сервере. Утилита `DISKPD`[1] может эмулировать стандартную рабочую нагрузку SQL Server.

Сеть

Мы не будем тратить много времени на обсуждение конфигурации сети в виртуализированных средах, потому что в большинстве случаев она диктуется физической инфраструктурой и конфигурацией на уровне хоста, и у вас мало возможностей ее контролировать. Но об одном аспекте я все же хотел бы упомянуть.

В виртуальных машинах очень легко создавать виртуальные сетевые адаптеры (`vNIC`), но они не гарантируют разделения трафика на физическом уровне. В зависимости от конфигурации трафик от разных виртуальных машин и/или виртуальных адаптеров может направляться через один и тот же физический сетевой адаптер на хосте. Чаще всего от этого страдает сеть проверки работоспособности в отказоустойчивом кластере Windows. Отсутствие физического разделения трафика может вызвать проблемы со стабильностью кластера и неожиданные аварийные переключения на хостах с виртуальными машинами с интенсивным сетевым трафиком.

Настройка виртуальной сети — это еще одна область, где вам необходимо тесно сотрудничать с коллегами из области виртуализации и инфраструктуры. Например, не стоит просто так запрашивать новые виртуальные адаптеры для вашей кластерной сети. Объясните, чего вы пытаетесь достичь, и попросите коллег разработать решение QoS, которое будет отдавать приоритет трафику пульса кластера.

[1] https://oreil.ly/r0M29

Управление виртуальными дисками

Как вы знаете, виртуализация абстрагирует конфигурацию хранилища от виртуальных машин. В виртуальной машине виртуальные диски отображаются как полностью подготовленные, но на уровне хоста их необходимо настроить одним из следующих способов.

Тонкая подготовка (thin provisioning)

В этом режиме виртуальный диск не занимает на хосте никакого места свыше того, которое используется в данный момент. При этом есть риск, что на хосте окажется выделено слишком много места и пространство исчерпается, если объем виртуальных дисков на нескольких виртуальных машинах превысит физическую емкость хранилища. В Hyper-V этот режим называется *динамическими дисками*, а в VMware — *тонкой подготовкой*.

Толстая подготовка без обнуления (thick provisioning without zeroing out)

На хосте заранее выделяется пространство для виртуальных дисков, но эти диски не обнуляются. Блоки на диске обнуляются при первой записи. Это позволяет быстро расширить диск за счет накладных расходов на первую запись. VMware называет этот режим zeroedthick provisioning.

Толстая подготовка с обнулением (thick provisioning with zeroing out)

Для виртуальных дисков на хосте заранее выделяется пространство, которое заполняется нулями при выделении. При этом нет накладных расходов на первую запись, однако расширение диска может занять некоторое время и породить нагрузку ввода/вывода. В Hyper-V этот режим называется *дисками фиксированного размера*, а в VMware — eagerzeroedthick provisioning («толстая подготовка с жадным обнулением»).

Решение о том, в каком режиме подготавливать диски для виртуальной машины, зависит от многих факторов. Например, в тонкой настройке нет ничего плохого, если накладные расходы на первую запись приемлемы и налажен эффективный мониторинг емкости хранилища. Однако при расширении виртуального диска виртуальная машина может на некоторое время «подвиснуть». При тонкой подготовке это время обычно измеряется миллисекундами, но если нужно обнулить большой объем пространства, это может занять секунды или даже минуты, что грозит проблемами или даже аварийным переключением в промышленной среде.

Обязательно включите управление дисковым пространством в свою стратегию обслуживания. Безопаснее расширять диски во время планового обслуживания системы и отключать автоматическое аварийное переключение, когда происходит расширение дисков на первичных узлах в кластере.

Стратегия и инструменты резервного копирования

Существует множество инструментов для резервного копирования на уровне виртуальных машин. Многие из них заявляют, что они совместимы с SQL Server и могут заменить собственные резервные копии SQL Server. Правда ли это и стоит ли их использовать — частая тема споров между специалистами по инфраструктуре и по базам данных. Инфраструктурщики любят эти инструменты, потому что они основаны на моментальных снимках, что экономит дисковое пространство. Специалисты по базам данных часто не доверяют им и не одобряют их ограничений.

Как и во многих других случаях, здесь все тоже зависит от обстоятельств. В некоторых ситуациях эти инструменты оказываются вполне приемлемыми, но каждый инструмент нужно тщательно анализировать, чтобы убедиться, что он соответствует вашим требованиям. В частности, нужно проверить такие возможности:

- Способность достижения целевых показателей точки восстановления (RPO) и целевого времени восстановления (RTO).
- Способность восстановления базы данных в новой системе с другой конфигурацией хранилища.
- Совместимость с такими функциями баз данных, как `FILESTREAM` и OLTP в памяти.
- Полноценная поддержка восстановления на определенный момент времени (а также желательно восстановление на уровне страниц — полезная функция для борьбы с повреждением базы данных).
- Автоматизированный процесс проверки резервной копии (который должен быть простым в реализации и использовании).
- Операционное воздействие: если инструмент может заблокировать работу VM, так ли он нужен?
- Конечно, не забудьте проверить, что инструмент работает правильно и выдержит аварийное восстановление (DR, Disaster Recovery) посреди резервного копирования. Также убедитесь, что инструмент сам не повреждает базу данных. Обычно я внедряю проверку согласованности базы данных и запускаю `DBCC CHECKDB` в ходе проверки резервной копии.

Не стоит сразу сбрасывать со счетов сторонние инструменты резервного копирования, сперва изучите их преимущества и недостатки. Эта информация поможет вам вести плодотворные обсуждения и принимать решения в сотрудничестве с другими командами.

Устранение неполадок в виртуальных средах

Исправлять проблемы и настраивать производительность экземпляров SQL Server в виртуальных средах можно примерно так же, как на «голом железе». Проблемы с экземплярами, базами данных и приложениями проявляются одинаково независимо от того, виртуализирован SQL Server или нет. Чтобы обнаруживать и устранять эти проблемы, можно применять одни и те же подходы во всех средах.

Но виртуализация добавляет к устранению неполадок еще один уровень сложности. В частности, приходится проверять, что конфигурация и производительность на уровне хоста не влияют на проблему. Что еще хуже, размещение виртуальных машин и их рабочая нагрузка редко бывают статичными, поэтому сегодняшний анализ может оказаться неактуальным завтра. К сожалению, очень немногие метрики видны внутри гостевых виртуальных машин. Чтобы обнаружить проблемы, нужно изучать хост.

Это еще одна причина привлекать администраторов виртуализации к устранению неполадок, особенно если необъяснимые проблемы с производительностью возникают внезапно без каких-либо изменений в рабочей нагрузке. Причиной этих проблем могут быть изменения на уровне хоста, а администраторы виртуализации умеют анализировать метрики на этом уровне. (Тем не менее помните, что возможны и другие причины: например, к тому же результату может привести деградация из-за сканирования параметров в планах выполнения, чувствительных к параметрам.)

Давайте обсудим, как устранять наиболее распространенные проблемы, связанные с виртуализацией.

Недостаточная пропускная способность процессора

Узкие места, связанные с ЦП, — это, пожалуй, самая сложная область для устранения неполадок в виртуализированных средах. Нужно анализировать как гостевую виртуальную машину, так и нагрузку хоста и устранять проблемы на обеих сторонах.

Информация о нагрузке ЦП в гостевой виртуальной машине — неполная. Гостевая машина отображает нагрузку виртуального ЦП в ОС, но не учитывает нагрузку на уровне хоста. Например, бывает, что гостевая ОС показывает низкую нагрузку ЦП, но система все равно страдает из-за узкого места ЦП, если сам хост перегружен.

Существует ключевая метрика, которая показывает, как долго виртуальная машина ожидает свободного ЦП для выполнения: она называется *временем готов-*

ности (ready time) ЦП в VMware и *временем ожидания (wait time)* ЦП в Hyper-V. Концептуально эти метрики похожи на ожидания `SOS_SCHEDULER_YIELD` в SQL Server, которые происходят, когда рабочий процесс задерживается в очереди выполнения, ожидая доступного планировщика.

Большое значение времени готовности ЦП указывает на проблемы с планированием на хосте. Это случается, когда хосту не хватает пропускной способности ЦП для обработки нагрузки. Проблему могут усугубить виртуальные машины с избыточным выделением ресурсов, потому что накладные расходы на планирование увеличиваются с количеством виртуальных ЦП.

Какое значение времени готовности процессора можно считать приемлемым? Этот вопрос тоже горячо обсуждается. Это время может варьироваться в зависимости от степени критичности виртуальных машин. В VMware меня устраивает, когда его доля составляет менее 1 % в системах уровня 1 и менее 2–3 % в системах уровня 2. Для Hyper-V задать пороговое значение немного сложнее, как я покажу в следующем разделе.

Если время готовности ЦП велико, проверьте конфигурацию виртуальной машины и нагрузку на хост. (Скоро я продемонстрирую, как узнать время готовности.) При необходимости переместите некоторые виртуальные машины на другие хосты. Не выделяйте для виртуальных машин избыточные ресурсы: экономичные машины, скорее всего, будут работать лучше, чем избыточно оснащенные. Однако не доходите до крайности и назначьте достаточное количество виртуальных ЦП для обработки нагрузки.

Давайте посмотрим, как получить метрики, связанные с процессором, в Hyper-V и VMware.

Анализ показателей ЦП в Hyper-V

На уровне хоста доступно несколько метрик производительности, связанных с ЦП. В Hyper-V это следующие показатели:

Объект производительности `Hyper-V Hypervisor Logical Processor`

Счетчики производительности в объекте `Hyper-V Hypervisor Logical Processor` предоставляют статистику о логических процессорах на хосте. Счетчик `% Total Run Time` показывает общее использование логических процессоров на хосте. Высокие значения (более 90 %) могут говорить о том, что хост перегружен. В этом случае попробуйте перенести некоторые виртуальные машины на другие хосты.

Счетчики `% Guest Run Time` и `% Hypervisor Run Time` показывают, сколько времени логический ЦП тратит на обработку кода гостевой ОС и гипервизора соответственно.

Счетчик производительности `Hyper-V Hypervisor Virtual Processor\% Total Run Time`

Этот счетчик показывает общее использование виртуальных ЦП в гостевой ОС. Высокие значения (выше 85 или 90 %) для всех виртуальных ЦП указывают на недостаточную пропускную способность ЦП в виртуальной машине. Если перегружены только некоторые виртуальные ЦП, это может свидетельствовать либо о несбалансированной рабочей нагрузке из-за неравномерного распределения планировщиков в узлах NUMA, либо о неэффективной конфигурации NUMA и/или ввода/вывода в Windows. Некоторые параметры ввода/вывода NUMA можно настроить — см. подробности в документации Microsoft[1].

Существуют также счетчики `% Guest Run Time` и `% Hypervisor Run Time`, которые похожи на показатели объекта производительности `Hyper-V Hypervisor Logical Processor`.

Счетчик производительности `Hyper-V Hypervisor Virtual Processor\CPU Wait Time Per Dispatch`

Этот счетчик показывает время ожидания ЦП (время готовности ЦП) в Hyper-V. К сожалению, он отображает не процент времени, в течение которого виртуальный ЦП находится в состоянии ожидания, а просто среднее время ожидания в наносекундах.

Трудно сказать, какие значения `CPU Wait Time Per Dispatch` можно считать признаком проблемы, потому что чем мощнее ЦП, тем этот счетчик будет ниже. Можно с уверенностью сказать, что в критических системах его значение не должно превышать нескольких микросекунд. Я также рекомендую принять некоторое значение счетчика за базовое и следить, как этот показатель изменяется со временем.

Наконец, обратите внимание на этот счетчик, если наблюдается высокое значение `Hyper-V Hypervisor Logical Processor\% Total Run Time`. Виртуализация предполагает, что хост будет работать под большой нагрузкой. Эта нагрузка не всегда отражает влияние на производительность виртуальных машин, и анализ метрики `CPU Wait Time Per Dispatch` помогает его оценить. Высокие значения обоих счетчиков указывают, что хост перегружен. Однако высокое значение `% Total Run Time` в сочетании с низким `CPU Wait Time Per Dispatch` можно считать приемлемым.

Анализ метрик ЦП в VMware ESXi

Существуют два подхода к оценке показателей производительности в VMware. Обзор всех метрик, доступных в VMware vSphere, выходит за рамки этой книги,

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

поэтому за дополнительной информацией обратитесь к документации VMware[1]. А здесь мы рассмотрим несколько наиболее важных показателей.

Во-первых, диаграммы производительности в клиенте vSphere позволяют в общих чертах анализировать использование ресурсов на хосте. Но значения на диаграммах усреднены по временны́м интервалам и не показывают всплесков и аномалий нагрузки.

Во-вторых, в vSphere есть утилита `ESXTOP`, которая в режиме реального времени показывает, что происходит на хосте. Она предоставляет крайне подробную информацию об использовании ресурсов и позволяет устранять проблемы с производительностью в реальном времени. Утилиту `ESXTOP` можно запустить из сеанса консоли или сеанса SSH на хосте. В ней есть несколько представлений, между которыми можно переключаться клавишами, соответствующими символам в скобках: например, ЦП (`c`), память (`m`), сеть (`n`) и статистика дисков виртуальной машины (`v`).

Также может быть полезно отфильтровать вывод, чтобы отображались только данные виртуальных машин. Для этого наберите заглавную букву (`V`). По умолчанию `ESXTOP` производит выборку данных каждые пять секунд. Чтобы изменить интервал, можно нажать (`s`) и ввести нужное количество секунд.

На рис. 15.5 показано представление ЦП в `ESXTOP`, собранное на одном из хостов.

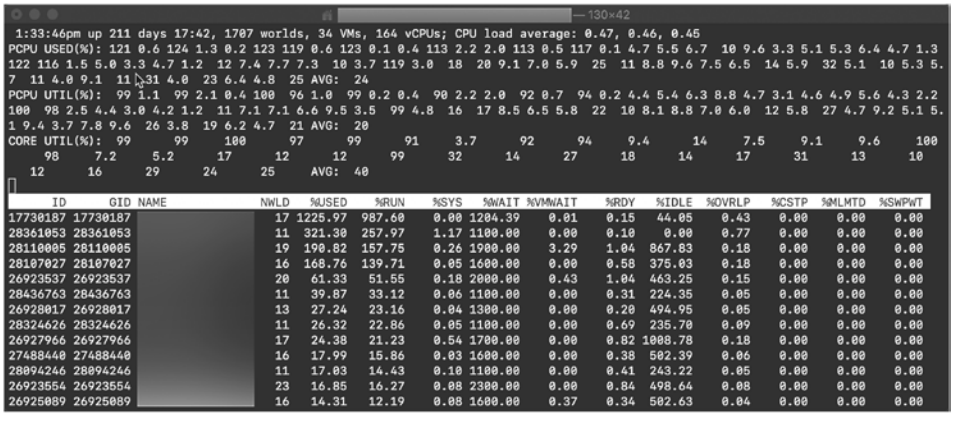


Рис. 15.5. `ESXTOP`: представление ЦП

В верхней строке экрана отображается статистика хоста. Три значения CPU Load Average отражают использование ЦП за последние 1, 5 и 15 минут соответственно. Значение 1.0 означает, что физические ЦП на хосте нагружены

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

полностью. Если CPU Load Average постоянно превышает 1.0, это значит, что хосту не хватает пропускной способности ЦП для обработки нагрузки.

Параметры PCPU USER(%) и PCPU UTIL(%) содержат информацию об использовании логического ядра (lCore), а CORE UTIL (%) — об использовании физического ядра (pCore). Эти метрики позволяют анализировать нагрузку ЦП хоста в режиме реального времени.

К сожалению, метрики использования ЦП для отдельных виртуальных машин труднее интерпретировать. ESXTOP вычисляет их на основе процентной доли базовой частоты ЦП на хосте, соответствующей различным виртуальным ЦП, которые назначены виртуальной машине. На расчеты также могут повлиять такие факторы, как гиперпоточность и Turbo Boost.

Тем не менее есть несколько показателей, на которые стоит обратить пристальное внимание.

Столбец %MLMTD

В этом столбце указан процент времени, в течение которого виртуальная машина была готова выполняться, но виртуальные ЦП не были запланированы из-за настроек пула ресурсов на хосте. Если вы видите значения выше 0, особенно на критически важных виртуальных машинах, обсудите выделение и лимитирование ресурсов с администратором виртуализации.

Столбец %RDY

В этом столбце показано время готовности ЦП для виртуальных машин, которое, как уже отмечалось, должно быть менее 1 % для критически важных виртуальных машин и не должно превышать 2–3 % для виртуальных машин уровней 2 и 3. Высокое время готовности ЦП может быть признаком перегруженного хоста, избыточного оснащения виртуальных машин или того и другого.

Время готовности ЦП также может накапливаться из-за лимитирования ресурсов, и в этом случае вы увидите ненулевые значения %MLMTD.

Столбец %CSTP

В этом столбце показано время совместной остановки виртуальной машины. В режиме облегченного совместного планирования VMware может приостановить виртуальную машину, если работа распределена неравномерно между виртуальными ЦП. Как и в случае с %RDY, значение %CSTP должно быть как можно меньше.

Время совместной остановки обычно не является проблемой для виртуальных машин с серверами баз данных, потому что нагрузка на все виртуальные ЦП сбалансирована. Если наблюдаются высокие значения %CSTP, проверьте планировщики и конфигурацию NUMA, а также убедитесь, что внутри виртуальной машины не запущены ненужные приложения.

Нехватка памяти

Вы уже знаете, что из-за перевыделения памяти на хосте драйвер накачки может вызвать нехватку памяти, что приведет к проблемам с производительностью и стабильностью SQL Server. Эта ситуация является распространенной причиной аварийного переключения в отказоустойчивых кластерах и группах доступности AlwaysOn, особенно если на узлах включена функция LPIM. SQL Server не высвобождает достаточно памяти, поэтому ОС перестает отвечать на запросы и не проходит проверку кластера `IsAlive`.

Эту проблему также можно обнаружить, проанализировав журналы ошибок отказоустойчивого кластера SQL Server и метрики накачанной памяти на уровне хоста и гостевой виртуальной машины. К сожалению, здесь нет простого решения, кроме того, чтобы лучше планировать мощность для хоста и гостевых виртуальных машин. Как и в случае с конфигурацией ЦП, не выделяйте виртуальным машинам слишком много памяти, если на узле недостаточно ресурсов.

Устранять проблемы с памятью на виртуальных машинах SQL Server можно с помощью методов, описанных в главе 7. В Hyper-V на уровне хоста можно посмотреть метрики `Memory\Available Mbytes` и `Hyper-V Dynamic Memory Balancer\Available Memory`. Их низкие значения указывают на нехватку памяти на хосте, что может привести к накачке памяти и другим проблемам.

Несколько метрик в утилите `ESXTOP` заслуживают особого внимания. Первая связана с подкачкой страниц виртуальной машины. Виртуальные машины с критически важными серверами баз данных не должны выполнять подкачку на уровне хоста. Столбец `%SWPWT` в представлении ЦП (рис. 15.5) показывает время, затраченное хостом на подкачку. Подробные метрики доступны в представлении памяти (рис 15.6), куда можно переключиться, нажав `m`. Существует несколько счетчиков памяти, и здесь показаны не все из них. Между ними можно переключаться, нажимая `f`.

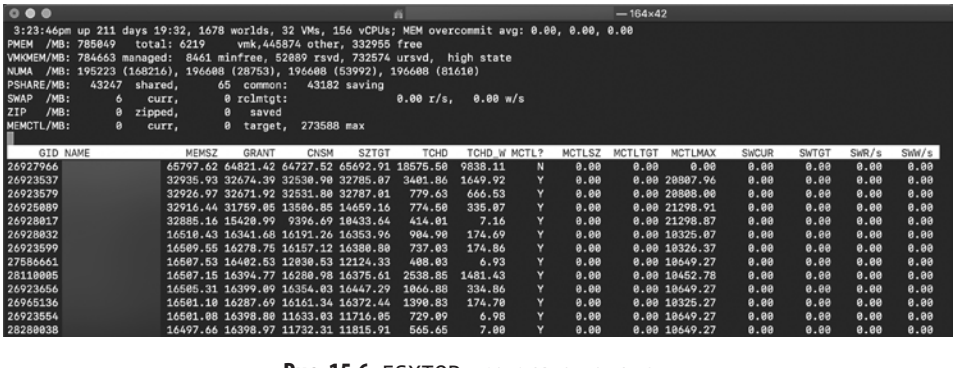


Рис. 15.6. ESXTOP: представление памяти

Давайте посмотрим на параметры, которые наиболее важны для устранения неполадок.

Столбцы, связанные с подкачкой страниц

Столбцы `SWR/s` и `SWW/s` отражают текущую активность подкачки виртуальной машины. Ненулевые значения указывают на то, что виртуальная машина в настоящее время выполняет подкачку. В виртуальных машинах баз данных не должно быть подкачки, потому что она ухудшает их производительность.

Столбец `SWCUR` показывает объем памяти виртуальной машины, которая в данный момент подкачана. Это значение тоже должно быть нулевым в виртуальных машинах баз данных, хотя можно наблюдать ненулевые значения, если некоторый неиспользуемый объем памяти виртуальной машины был подкачан какое-то время назад и с тех пор не использовался. Низкое значение в столбце `SWCUR` и отсутствие подкачки в столбцах `SWR/s` и `SWW/s` может быть приемлемым в некритических системах.

Важно помнить, что эти столбцы указывают на подкачку на уровне хоста и не видны внутри виртуальной машины.

Накачка памяти

Есть несколько столбцов `MCTL`, связанных с накачкой. Столбец `MCTL?` указывает, установлен ли драйвер накачки в ОС. Столбцы `MCTLZ (MB)` и `MCTLTGT (MB)` отображают объем отозванной памяти и объем памяти, которую хост хочет отозвать. Наконец, столбец `MCTLMAX (MB)` показывает максимальный объем памяти, который может отозвать драйвер накачки.

Опять же, накачка чрезвычайно опасна для виртуальных машин баз данных. Чтобы избежать проблем, правильно настройте виртуальные машины и хосты, а также не перевыделяйте память.

Производительность дисковой подсистемы

Как вы помните из главы 3, для устранения неполадок дисковой подсистемы нужно просмотреть весь стек хранилища от SQL Server до физического массива хранения, проанализировать метрики каждого компонента и выявить узкие места.

Виртуализация — это просто еще один уровень в этом стеке, и следует убедиться, что он не приводит к заметным накладным расходам на производительность ввода/вывода. В Hyper-V можно сравнивать счетчики производительности дисков хостовой и гостевой ОС. В VMware можно использовать информацию утилиты `ESXTOP`.

На рис. 15.7 показано представление виртуального диска виртуальной машины в утилите `ESXTOP`. Столбцы отображают статистику операций ввода/вывода и задержки для виртуальных дисков.

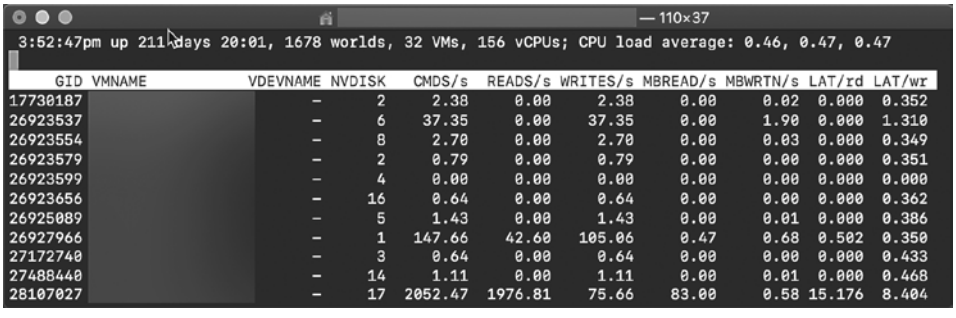


Рис. 15.7. ESXTOP: представление виртуального диска

Также можно изучить представление дискового адаптера (для этого нажмите `d`), показанное на рис. 15.8. Здесь можно увидеть, насколько операции ввода/вывода сбалансированы между дисковыми адаптерами, что поможет устранить проблемы с высокой задержкой виртуального диска. Столбцы `DAVG`, `KAVG`, `QAVG` и `GAVG` представляют устройства, ядро VMware, очереди адаптера и задержки ОС виртуальных машин для запросов ввода/вывода соответственно.

Убедитесь, что в гостевых виртуальных машинах используются последние версии инструментов виртуализации и что все драйверы хоста и гостевой системы обновлены. Разделите рабочую нагрузку между несколькими адаптерами `vSCSI`. При необходимости увеличьте глубину очередей.

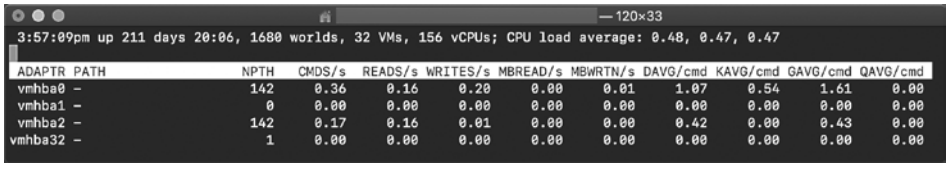


Рис. 15.8. ESXTOP: представление дискового адаптера

Убедитесь, что в виртуальных машинах нет моментальных снимков (snapshots). Они добавляют значительные накладные расходы ввода/вывода, которые могут повлиять на высоконагруженные системы. Кроме того, некоторые задачи обслуживания, такие как клонирование виртуальных машин или перенос больших виртуальных дисков в другое хранилище, могут вызвать внезапное и труднообъяснимое падение производительности виртуальных машин.

Нет сомнений, что виртуализация усложняет устранение неполадок и может привести к проблемам, но их обычно удается решить. Большинство проблем,

с которыми я сталкиваюсь в виртуальных средах, связано с неправильной настройкой виртуализации или неэффективной стратегией устранения неполадок, которая не учитывает экосистему в целом. Здесь помогает целостный подход и сотрудничество между специалистами разного профиля.

Бывают случаи, когда нужно получить максимальную отдачу от оборудования, и тогда виртуализация — не лучший выбор. За гибкость, которую она обеспечивает, вы расплачиваетесь накладными расходами. Помните об этом, принимая решения.

Резюме

Виртуализация стала чрезвычайно распространенным явлением. Она снижает стоимость оборудования и упрощает обслуживание. Но она также добавляет накладные расходы, которые увеличиваются с размером виртуальных машин, хотя во многих случаях оказываются приемлемыми.

Правильное планирование мощности крайне важно для виртуализации. Не перевыделяйте ресурсы на узлах с критически важными виртуальными машинами баз данных уровня 1. Используйте резервирование ресурсов, когда это возможно. Вместе с тем виртуальные машины уровней 2 и 3 могут выдерживать некоторую степень перевыделения.

Как правило, не следует выделять виртуальным машинам избыточные ресурсы, особенно в средах с перевыделением. В меньших виртуальных машинах легче настраивать планирование и управление ресурсами, и они часто обеспечивают более высокую производительность, чем более крупные.

Обращайте внимание на согласование ЦП, памяти и NUMA в конфигурации виртуальной машины. Убедитесь, что планировщики равномерно распределены по узлам NUMA в SQL Server. Правильно настройте конфигурацию `vSocket/vCore`, особенно если вы используете выпуски SQL Server, отличные от Enterprise.

Хотя неполадки производительности SQL Server в виртуальной среде устраняются примерно так же, как в обычной, принимайте в расчет уровень виртуализации. Обращайте внимание на время готовности ЦП, накачку памяти и дополнительные накладные расходы, которые могут быть вызваны неправильно настроенной виртуализацией. Устраняя неполадки, сотрудничайте с администраторами виртуализации.

В следующей главе мы поговорим об SQL Server в облачных средах и о проблемах, которые могут возникнуть в облачной конфигурации.

Чек-лист устранения неполадок

- ☐ Проверить конфигурацию виртуальной машины, уделяя особое внимание конфигурации ЦП и памяти, а также согласованию NUMA.
- ☐ Убедиться, что планировщики равномерно распределены по узлам NUMA в SQL Server.
- ☐ Убедиться, что установлены последние версии инструментов и драйверов гостевой ОС.
- ☐ Разделить рабочую нагрузку ввода/вывода между несколькими контроллерами `vSCSI` и при необходимости увеличить глубину очереди.
- ☐ Проверить конфигурацию хоста. Убедиться, что хост не перевыделен, если на нем работают критически важные системы уровня 1.
- ☐ Не выделять виртуальной машине SQL Server избыточные ресурсы, если в ней не используется резервирование ресурсов и нет выделенных ресурсов на хосте.
- ☐ Настроить параметры памяти SQL Server и LPIM в соответствии с вашей средой.
- ☐ Убедиться, что на промышленных виртуальных машинах нет моментальных снимков.

ГЛАВА 16

SQL Server в облаке

Облачные вычисления — уже давно не экзотика. Компании сокращают площадь дата-центров и либо создают гибридные решения, либо полностью переходят в облако. Таким образом, устранение неполадок и настройка облачных систем баз данных стали обычной задачей для специалистов по БД.

Экземпляры облачных баз данных функционируют на той же платформе SQL Server, и при работе с ними можно использовать знакомые вам инструменты и приемы. Но есть и небольшие различия, которые мы рассмотрим в этой главе.

Мы начнем с высокоуровнего обзора облачных приложений, а затем перейдем к облачным возможностям SQL Server. Сперва пойдет речь об экземплярах SQL Server, работающих на облачных виртуальных машинах, а затем об управляемых службах баз данных, которые доступны в Microsoft Azure, Amazon AWS и Google Cloud Platform. Я также в общих чертах опишу архитектуру служб и инструменты мониторинга платформ и расскажу об их ограничениях.

Облачные платформы с высоты птичьего полета

Давным-давно, когда облачные вычисления только зарождались, на одной конференции мне попалась наклейка (рис. 16.1). На ней было написано: «Облака не существует, это просто чей-то чужой компьютер».

Рис. 16.1. Облака не существует

Тогда это можно было считать лучшим определением облачных вычислений, однако со временем все стало сложнее. Сейчас облака — это по-прежнему *чьи-то чужие дата-центры*, но они эволюционировали и теперь предоставляют широкий ассортимент возможностей: помогают создавать приложения, предлагают готовые решения сложных проблем и зачастую значительно ускоряют реализацию проектов.

Все основные поставщики облачных служб предлагают услуги управляемых баз данных, принимая на себя рутинные задачи по их администрированию и обеспечивая высокую доступность баз данных. Эти службы также позволяют оснащать и запускать обычные виртуальные машины и переносить локальную инфраструктуру в облако.

На мой взгляд, подход «взять и перенести» редко приводит к лучшим долгосрочным результатам. В некоторых случаях это оправданно, и часто это самый быстрый способ миграции сложных локальных систем. Тем не менее я искренне считаю, что если вы хотите получить максимальную отдачу от облака, то систему нужно специально проектировать для работы в облаке. Только тогда вам удастся использовать преимущества облачных служб и учесть особенности облачных платформ.

Существует несколько ключевых различий между облачной и локальной инфраструктурами.

Надежность платформы

Любое облако работает на потребительском оборудовании. Сбои в облаках происходят чаще, чем на локальном оборудовании, поэтому приложения и серверы баз данных должны уметь справляться с отказами.

Сразу приведу пример. У Amazon AWS есть соглашение об уровне обслуживания (SLA), предусматривающее 99,5 % времени бесперебойной работы для виртуальных машин EC2, что означает до 3,6 часа простоя в месяц. Что гораздо хуже, в случае нарушения SLA единственное, что вы получите обратно, — это компенсацию за обслуживание, которая не покроет убытки бизнеса, возникшие из-за простоя.

Вопросы надежности платформы выходят за рамки уровня базы данных. Приложения должны быть устойчивыми и уметь обрабатывать случайные ошибки. Они также должны быть идемпотентными. Например, если база данных или очередь сообщений становится недоступной посреди бизнес-транзакции, у системы должна быть возможность повторно обработать транзакцию после восстановления.

Лимитирование ресурсов

Второй ключевой фактор — лимитирование ресурсов (throttling). Облачные сервисы и ресурсы применяют жесткое лимитирование. Например, если вы платите за тома эластичного блочного хранилища (EBS, elastic block storage)

с определенным количеством операций ввода/вывода в секунду (IOPS) в AWS, вы не получите ни на одну IOPS больше, чем оплатили. В локальной виртуализации ситуация другая: на критические виртуальные машины могут влиять «шумные соседи» и общая нагрузка хоста, но к ним редко применяется лимитирование со стороны QoS.

Наличие лимитирования означает, что при оснащении облачных систем нужно тщательно планировать мощность. К счастью, последствия ошибок и недостаточно выделенной мощности менее серьезны, чем при неподходящей мощности локального оборудования. В большинстве случаев облачные сервисы можно быстро масштабировать, хотя иногда это влетает в копеечку.

К сожалению, некоторые ограничения оказываются *очень* жесткими. Например, у виртуальных машин задается предельный уровень трафика ввода/вывода и сети, службы управляемых баз данных могут ограничивать скорость пополнения журнала транзакций и т. д. Проектируя облачную инфраструктуру, необходимо учитывать эти ограничения. Облако может поддерживать почти любые рабочие нагрузки, однако, если вы перемещаете в облако высоконагруженную систему, ее почти наверняка лучше перепроектировать.

Топология

Я уверен, что вы знакомы с облачной топологией, но тем не менее перечислю ее ключевые элементы.

Регион

Каждый облачный регион соответствует отдельному дата-центру в определенном географическом регионе. Например, дата-центр Microsoft Azure West US находится в Калифорнии, а West US 2 — в штате Вашингтон.

Распределение служб и баз данных по регионам позволяет создавать геоизбыточные системы. Однако при проектировании систем следует помнить о задержке межрегиональной связи. В крайних случаях, когда вы распределяете серверы по разным континентам, задержка может достигать сотен миллисекунд. Из-за межрегиональных обращений к базам данных может значительно увеличиться продолжительность транзакций. По возможности лучше реализовать асинхронную связь на основе сообщений.

Зоны доступности

Зоны доступности (AZ, availability zones) — это изолированные участки инфраструктуры в каждом регионе. Их можно представить как отдельные здания одного дата-центра или даже как отдельные дата-центры в одном городе, независимые друг от друга. Они обеспечивают необходимую избыточность в пределах каждого региона: проблемы в одной зоне доступности *теоретически* не должны затрагивать другие зоны.

Поскольку зоны доступности разделены физически, у них разная сетевая инфраструктура и связь между ними происходит с задержкой. Эта задержка обычно очень мала — менее 1–2 мс, — но все же может увеличить нагрузку на синхронную репликацию групп доступности в высоконагруженных системах OLTP.

Большинство управляемых облачных служб «из коробки» обеспечивают избыточное присутствие в различных зонах доступности или могут быть настроены так, чтобы работать в нескольких зонах. В то же время каждая облачная виртуальная машина находится только в одной зоне доступности, поэтому для облачной виртуализации необходимо внедрять эффективные решения высокой доступности.

Прежде чем мы перейдем к вопросам оснащения облачного SQL Server и устранения соответствующих неполадок, я хотел бы кратко затронуть тему подключения к экземплярам облачной базы данных.

Связь и обработка случайных ошибок

Проблемы со связью — один из самых распространенных источников ошибок при развертывании облачных систем. К счастью, чаще всего с этими ошибками относительно легко справиться. Проблемы со связью бывают двух типов: отсутствие доступа к экземпляру базы данных и случайные ошибки.

Доступ к экземпляру базы данных

Первая проблема — отсутствие доступа к экземпляру базы данных. При подготовке облачного ресурса его возможности связи по умолчанию ограниченны. Вы не можете подключиться к нему извне, и другие облачные службы тоже не могут, пока не настроены соответствующие правила доступа. Эта ситуация будет проявляться как общая ошибка подключения, когда клиент не может подключиться к экземпляру SQL Server.

Решение проблемы существенно зависит от топологии. Если вам нужно подключиться к базе данных через общедоступный интернет, откройте для базы публичный доступ и разрешите IP-адреса клиентов. В более сложных случаях, таких как подключение облачных ресурсов к локальной инфраструктуре, скооперируйтесь со специалистами по облачным и сетевым технологиям и вместе с ними добейтесь, чтобы экземпляр базы данных был включен в нужную виртуальную сеть или виртуальное частное облако (VPC, virtual private cloud) и чтобы группы безопасности (правила сетевого брандмауэра) были правильно настроены.

Случайные ошибки

Второй тип проблем связан со случайными ошибками. Помимо состояния аварийного переключения, сервер базы данных может не позволить подключиться к нему, если исчерпаны ограничения тарифного плана. Например, сервер может принудительно завершать запросы или даже соединения с интенсивным использованием ресурсов, если сеанс создает слишком много записей журнала транзакций.

Когда возникают случайные ошибки, то управляемые службы баз данных Microsoft Azure предоставляют дополнительные сведения; коды и описания ошибок можно найти в документации Microsoft[1]. Вместе с тем управляемые службы Amazon Web Services (AWS) и Google Cloud Platform (GCP) не обеспечивают настолько информативных сообщений об ошибках, и неполадки могут проявляться в виде нехватки доступных исполнителей (ожидания `THREADPOOL`) или общих проблем с производительностью.

Поскольку приложение должно быть устойчивым, оно должно реализовывать логику повторных попыток при обработке ошибок. По возможности перед повторной попыткой лучше открыть соединение заново, потому что из-за некоторых случайных ошибок сеанс может стать неработоспособным. Кроме того, в облачных службах предусмотрены меры по предотвращению DoS-атак, поэтому при повторном открытии соединения не заваливайте экземпляр запросами на подключение. Если не удается повторно подключиться немедленно, подождите несколько секунд перед новой попыткой.

Предостережение: избегайте конфигураций «клиент — сервер», где удаленные клиентские приложения обращаются к экземплярам облачной базы данных. Такие конфигурации подвержены высоким задержкам, случайным ошибкам и проблемам с подключением. Они могут быть приемлемы для некритичных приложений, но недостаточно устойчивы для использования в критических системах. В таких системах ожидания `ASYNC_NETWORK_IO` обычно не покидают вершины списка ожиданий из-за задержек между клиентом и сервером. К сожалению, рабочие потоки, ожидающие в этом состоянии, недоступны для других сеансов, что может привести к нехватке рабочих потоков и ожиданиям `THREADPOOL`. Обращайте внимание на эти ожидания при устранении неполадок в облачных системах.

Наконец, не забудьте увеличить значение времени ожидания соединения и команды, когда вы подключаетесь к SQL Server удаленно, особенно если вы планируете запускать клиентские приложения в медленных сетях.

Теперь давайте рассмотрим варианты подготовки SQL Server в облаке.

---

[1] https://oreil.ly/WMF71

SQL Server в облачных виртуальных машинах

Каждый поставщик облачных услуг предлагает свои управляемые службы SQL Server, но некоторые команды предпочитают сами управлять экземплярами и размещать SQL Server на облачных виртуальных машинах. Это обычно происходит в таких случаях:

- Миграции в стиле «взять и перенести», когда вся инфраструктура перемещается в облако «как есть» с минимальными изменениями.
- Сложные требования к инфраструктуре SQL Server или рабочие нагрузки, которые не поддерживаются управляемыми службами.
- Ситуации, когда специалисты, работающие с базами данных, хотят иметь полный контроль над средой.
- Проприетарные реализации баз данных как услуги, которые могут оказаться значительно дешевле, чем управляемые службы с профессиональными командами сопровождения.

Поддерживать экземпляры SQL Server на облачных виртуальных машинах и устранять их неполадки можно так же, как в случае с *виртуализированными* локальными экземплярами, о которых говорилось в главе 15. По сути, в обоих случаях вы имеете дело с одним и тем же продуктом и одним и тем же набором инструментов. Облачные порталы позволяют получить метрики использования ресурсов на уровне гипервизора, что тоже концептуально похоже на локальную виртуальную машину.

Тем не менее в поведении платформ есть несколько различий, на которые нужно обратить внимание. Давайте рассмотрим их подробнее.

Настройка ввода/вывода и производительность

Современное аппаратное обеспечение позволяет создавать очень мощные серверы, которые могут скрывать проблемы с производительностью. Например, влияние неоптимизированных запросов, чрезмерное протоколирование и другие проблемы компенсируются дополнительными ЦП, большим объемом ОЗУ для кэширования данных и хранилищем с задержкой менее миллисекунды. Однако иногда эти проблемы всплывают вновь при перемещении SQL Server в облако.

Во время планирования мощности легко оценить требования к вычислительным ресурсам (ЦП и ОЗУ) виртуальных машин, потому что производительность виртуальных машин одинакового размера в локальной среде и в облаке сопоставима. Но в облаке нужно обращать внимание на классы экземпляров виртуальных машин и поколение оборудования, потому что они определяют базовое оснащение.

Хранилище — отдельная тема. В локальной среде масштабируемость хранилища обычно зависит от оборудования. Современные хранилища могут выдерживать нагрузку в сотни тысяч операций ввода/вывода в секунду, обладают пропускной способностью в несколько гигабайт в секунду и могут масштабироваться при всплесках рабочей нагрузки SQL Server.

Облачное хранилище устроено иначе. Каждый облачный поставщик поддерживает несколько типов и уровней хранилища и ограничивает IOPS и пропускную способность каждого из них. Некоторые уровни хранилищ позволяют превысить ограничения в случае всплесков нагрузки, но потом сильно лимитируют мощность.

Что еще хуже, быстрые хранилища и большие виртуальные машины стоят дорого, поэтому часто на виртуальную инфраструктуру выделяется гораздо меньше ресурсов по сравнению с локальной. Виртуальные машины с медленным вводом/выводом и недостаточным размером могут столкнуться с проблемами производительности, которые не были заметны на быстром локальным оборудовании.

Конечно, лучше устранять первопричины проблем и налаживать систему. Однако, если это невозможно или нецелесообразно, убедитесь, что ваша облачная инфраструктура, а особенно хранилище, должным образом подготовлена. Я не могу дать универсального совета, который подходил бы ко всем системам, но предложу несколько рекомендаций.

- Перед миграцией в облако проанализируйте пропускную способность ввода/вывода в своих локальных системах. В качестве целевых показателей облачной конфигурации используйте ключевые индикаторы производительности (KPI) на уровне хранилища.
- Используйте типы дисков с гарантированными характеристиками производительности, способные обрабатывать вашу нагрузку.
- Размещайте файлы данных и журналов на разных дисках, чтобы по мере необходимости масштабировать пропускную способность. То же самое относится к различным файловым группам и в особых случаях к файлам данных.
- Попробуйте настроить в своей ОС RAID-массивы. Например, в Microsoft Azure два диска премиум-класса P30 емкостью 1 Тбайт в массиве RAID-0 обеспечат больше IOPS и лучшую пропускную способность, чем один диск P40 емкостью 2 Тбайт.
- Помните, что виртуальные машины ограничивают количество операций ввода/вывода в секунду и пропускную способность.

Чтобы устранять неполадки с производительностью диска, можно использовать те же подходы, которые рассматривались в главе 3. Также можно изучить метрики ввода/вывода и оценить, насколько вы приблизились к пределам использования ресурсов облачного портала.

Настройка высокой доступности

Грамотная реализация высокой доступности — *обязательное требование* для SQL Server, работающего на облачной виртуальной машине. Обычно для этого специалисты по базам данных внедряют группы доступности, но можно использовать и отказоустойчивые кластеры.

Это особенно важно при подготовке виртуальных машин в разных зонах доступности для избыточности. Нужно будет создать и настроить отказоустойчивый кластер Windows Server (WSFC) с несколькими подсетями. Это приведет к небольшой, обычно приемлемой задержке синхронной репликации.

Если инфраструктура SQL Server должна распространяться на несколько регионов, никогда не используйте межрегиональную синхронную репликацию. Сетевая задержка приведет к чрезвычайно высоким ожиданиям `HADR_SYNC_COMMIT` и может создать масштабное блокирование в базе данных.

Если стратегия высокой доступности в вашей организации требует, чтобы система могла выдержать аварию на уровне региона и продолжать работу во вспомогательном регионе после аварийного переключения, то во вспомогательном регионе имеет смысл подготовить несколько реплик. Для этого обычно лучше использовать распределенные группы доступности, чем отдельные реплики. Они позволяют сократить сетевой трафик между регионами и, на мой взгляд, ими проще управлять с точки зрения высокой доступности. Имейте в виду, что в случае аварии на уровне региона для распределенных групп доступности потребуется другая стратегия восстановления после аварийного переключения по сравнению с обычными группами доступности.

Попробуйте использовать облачную службу DNS (Azure DNS, Amazon Route53, Google Cloud DNS) для подключения приложений. Это упростит аварийное переключение между регионами, особенно если у вас настроены распределенные группы доступности. Также может принести пользу архитектура, в которой применяются отдельные пулы соединений для первичных и доступных для чтения вторичных реплик в мультирегиональных приложениях типа «активный — активный». Службы в каждом регионе будут подключаться к локальным репликам SQL Server и выполнять запросы только для чтения, когда допустимы задержка асинхронной репликации и доступ к слегка устаревшим данным.

Межрегиональная задержка

Облачная инфраструктура практически провоцирует создавать геоизбыточные системы с приложениями, которые обслуживают запросы клиентов в нескольких регионах. Но это не такая уж простая задача, если вы используете реляционную базу данных, которая выполняет операции записи на одном первичном узле. Проблема — в задержке межрегиональных вызовов.

Эта задержка зависит от расстояния между дата-центрами. Если вы распределили серверы между регионами Azure East US и East US2, задержка вряд ли превысит несколько миллисекунд, но когда ваши дата-центры находятся на разных концах земного шара, задержка каждого вызова может достигать 200–300 мс.

Спроектируйте систему так, чтобы уменьшить влияние задержек. Этому могут помешать приложения, которые выполняют много отдельных запросов в бизнес-транзакциях. Суммарная продолжительность вызовов быстро увеличивается, что приводит к проблемам с производительностью и ухудшению пользовательских качеств.

Для примера представьте, что приложению, расположенному в регионе Azure West US, нужно вставить 100 строк в базу данных, размещенную в дата-центре в Западной Европе. Реализация со 100 отдельными вызовами `INSERT` займет около 15 секунд и может привести к ощутимому блокированию, потому что монопольные блокировки (`X`) будут удерживаться в течение длительного времени. Вместо этого можно передать хранимой процедуре все 100 строк в одном пакете параметров с табличным значением (TVP), и получится всего один межрегиональный вызов, что намного эффективнее.

Некоторые проблемы с задержкой можно решить, если использовать для нагрузки чтения локальные доступные для чтения вторичные узлы в регионе. Но помните, что репликация в группах доступности тоже подвержена задержкам. Не используйте доступные для чтения вторичные реплики, если вам нужно прочитать свежие данные. И, что еще важнее, не смешивайте в одной бизнес-транзакции операции чтения из вторичных узлов и операции записи в первичный узел.

А что если приложение вставляет данные в первичный узел и сразу перезагружает их из реплики, ожидая, что они будут обновлены? Это не сработает: проблема в задержке, связанной с межрегиональной репликацией. В одном регионе так тоже не стоит делать: эта система может сработать в обычной ситуации, но любые всплески протоколирования или заторы в очередях отправки и повтора могут всё сломать.

Наконец, не забывайте о задержке вызовов аутентификации SQL Server к контроллеру домена. В главе 13 мы узнали, что эти вызовы могут ощутимо влиять на систему, если их задержка велика. Если вам нужно использовать аутентификацию Windows, попробуйте в каждом регионе обеспечить контроллер домена или использовать облачные службы, которые интегрируются с Active Directory.

И все же в остальном запуск SQL Server на облачных виртуальных машинах очень похож на запуск виртуализированных экземпляров SQL Server в локальной среде. Для управления, настройки и устранения неполадок применяется один и тот же набор инструментов и методов, которые одинаково работают со всеми облачными поставщиками.

Теперь посмотрим, какие аспекты служб управляемых баз данных различаются у разных поставщиков.

Управляемые службы Microsoft Azure SQL

Помимо SQL Server, работающего на виртуальных машинах Azure, семейство Microsoft Azure SQL включает базы данных Azure SQL, управляемые экземпляры Azure SQL и Azure SQL для периферийных вычислений (версия SQL Server, оптимизированная для интернета вещей (IoT) и работающая на периферийных устройствах на базе ARM и x86). В этом разделе я сосредоточусь на управляемых экземплярах и базах данных Azure SQL.

На внутреннем уровне все эти технологии работают на одном и том же ядре SQL Server, используют одну и ту же кодовую базу и ведут себя одинаково. Методы настройки и устранения неполадок, описанные в этой книге, будут работать во всех этих системах. Правда, в некоторых случаях, возможно, будут различаться методы сбора данных. Например, управляемые облачные службы абстрагируют вас от нижележащей ОС, поэтому, чтобы изучить потребление ресурсов, вместо использования утилиты `PerfMon` нужно заходить на портал Azure или запрашивать динамические административные представления. Однако подходы к анализу и устранению неполадок не зависят от того, какую технологию вы используете.

Рекомендации по архитектуре и проектированию служб

Microsoft представила базы данных Azure SQL в 2010 году. Их можно рассматривать как «БД как услугу» (DBaaS), предоставляющую вам базу данных SQL Server. Эта технология поддерживает большинство возможностей баз данных и обеспечивает высокую доступность, резервное копирование и обслуживание SQL Server и нижележащей ОС. Однако у нее довольно много ограничений, особенно на уровне экземпляра. Например, нельзя выполнять запросы между базами данных, использовать CLR, брокер служб (Service Broker) или SQL Agent.

В 2018 году Microsoft добавила в эту линейку новый продукт — управляемые экземпляры Azure SQL (SQL MI, SQL Managed Instances). Если говорить коротко, то SQL MI — это виртуализированный экземпляр SQL Server, который поддерживает большинство стандартных функций SQL Server. Чаще всего локальные экземпляры SQL Server можно без особых усилий перемещать на SQL MI.

Не забывайте учитывать ограничения ресурсов разных технологий. Например, ограничения пропускной способности ввода/вывода и скорости пополнения журналов могут быть слишком строгими для рабочей нагрузки высокопроизводительной системы OLTP.

Как и базы данных Azure SQL, SQL MI — это управляемый облачный сервис, который берет на себя задачи резервного копирования, высокой доступности и обновлений ПО. В обеих технологиях вам по-прежнему придется выполнять административные задачи, такие как обслуживание индексов и статистики.

Мощность базового оборудования можно понижать или повышать, указав количество логических ЦП (виртуальных ядер) для того или иного экземпляра. От количества виртуальных ядер также зависит объем памяти. Например, при использовании SQL MI каждое виртуальное ядро предоставляет вам 5,1, 7 или 13,6 Гбайт ОЗУ в зависимости от типа экземпляра. (Учтите, что по мере появления новых типов оборудования эти цифры могут меняться.)

У обеих технологий есть несколько уровней обслуживания. Уровни *General Purpose* и *Business Critical* поддерживаются как в базах данных, так и в управляемых экземплярах Azure SQL. Третий уровень, *Hyperscale*, доступен только в базах данных Azure SQL.

От уровня сервиса зависит соглашение об уровне обслуживания (SLA) и доступность некоторых функций: например, In-Memory OLTP доступна только на уровне Business Critical. Однако более важными характеристиками уровня являются хранилище и реализация высокой доступности.

Базы данных Azure SQL поддерживают другую модель покупки, которая основана на единицах пропускной способности базы данных (DTU, Database Throughput Units) и поставляется с тремя другими уровнями обслуживания. Уровни Basic и Standard похожи на General Purpose, уровень Premium аналогичен Business Critical.

На рис. 16.2 показана конфигурация вычислительных ресурсов и хранилища на уровне General Purpose. Слой вычислений с процессом SQL Server не сохраняет состояния. Он содержит только временные данные, такие как `tempdb`, в локальном хранилище SSD. Файлы базы данных и резервные копии хранятся в избыточном хранилище Azure Blob.

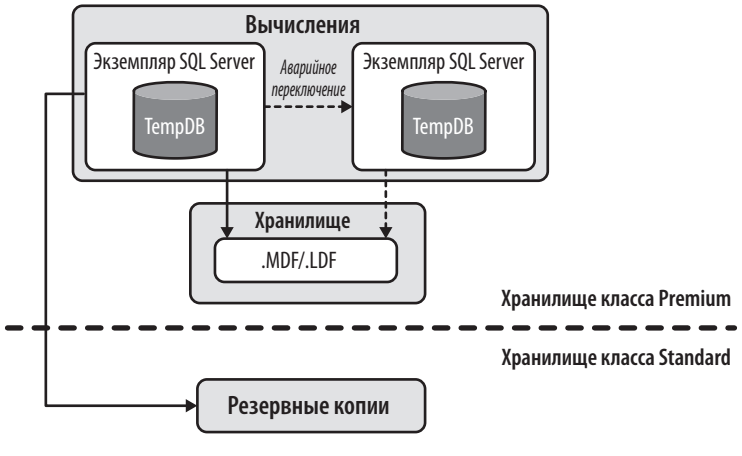


Рис. 16.2. Конфигурация вычислительных ресурсов и хранилища на уровне обслуживания General Purpose

В случае аварийного переключения другой экземпляр SQL Server подключается к тем же файлам базы данных и берет на себя рабочую нагрузку. Концептуально он ведет себя так же, как отказоустойчивый кластер AlwaysOn SQL Server, но внутренняя реализация отличается и работает на Azure Service Fabric.

В базе данных Azure SQL можно настроить избыточность по зонам, когда хранилище реплицируется в нескольких зонах доступности. В управляемых экземплярах Azure SQL на момент написания этого раздела такая функция еще не поддерживалась.

На уровне Business Critical реализация основана на группах доступности AlwaysOn. В конфигурации, показанной на рис. 16.3, база данных или управляемый экземпляр Azure SQL состоит из нескольких реплик, в которых файлы базы данных хранятся на локальных накопителях SSD. Аварийное переключение в этой конфигурации — это просто обычное аварийное переключение группы доступности AlwaysOn.

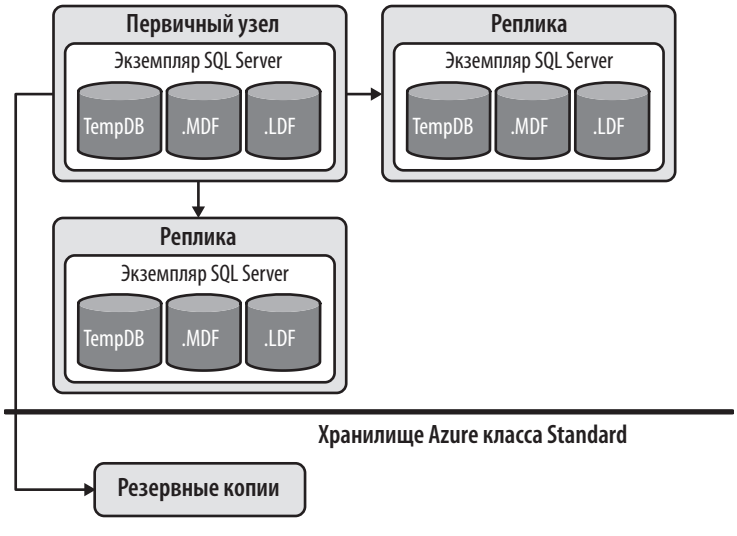


Рис. 16.3. Конфигурация вычислительных ресурсов и хранилища на уровне обслуживания Business Critical

Нетрудно догадаться, что от нижележащей топологии зависят некоторые характеристики системы. Например, диск на уровне General Purpose работает медленнее, но позволяет эксплуатировать большие базы данных, потому что не ограничен объемом локальных SSD. Вместе с тем уровень Business Critical обеспечивает меньшую задержку ввода/вывода и позволяет масштабировать рабочую нагрузку чтения с помощью доступных для чтения вторичных реплик. Но взамен вы получаете накладные расходы на синхронную репликацию и по-

тенциальные проблемы с ожиданиями `HADR_SYNC_COMMIT` (об этом мы говорили в главе 12).

Как и уровень General Purpose, Business Critical может быть избыточен по зонам. В этом режиме реплики распределяются по зонам доступности. При этом обеспечивается более высокая доступность за счет небольших накладных расходов во время синхронной репликации.

Уровень сервиса Hyperscale поддерживает очень большие базы данных (VLDB, very large databases). Его архитектура (рис. 16.4) устроена сложнее и содержит следующие компоненты:

- Серверы страниц, которые управляют файлами данных, хранящимися в Azure Blob.
- Вычислительные узлы на основе групп доступности, обрабатывающие запросы.
- Систему управления журналом транзакций Log Service, которая принимает поток записей журнала от первичного узла и распределяет его по серверам страниц и вторичным вычислительным узлам.

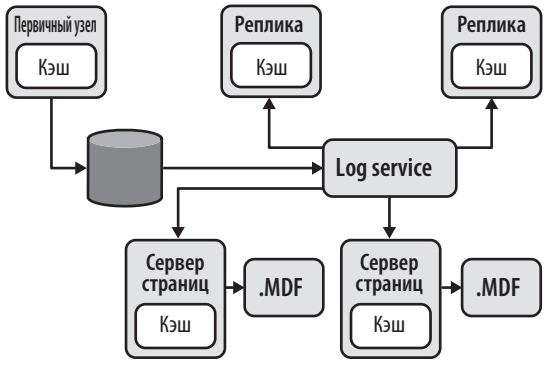


Рис. 16.4. Архитектура уровня обслуживания Hyperscale

По состоянию на декабрь 2021 года уровень Hyperscale поддерживает базы данных размером до 100 Тбайт. В теории серверы страниц позволяют масштабировать базы данных практически неограниченно, но их производительность сильно зависит от рабочей нагрузки.

У каждого вычислительного узла есть локальный кэш на основе SSD для *горячих* (активных) данных, опирающийся на технологию Buffer Pool Extension. Когда активные данные помещаются в кэш, узлы могут очень быстро получать к ним доступ. Но получение данных с удаленных серверов страниц происходит гораздо менее эффективно.

Протестируйте свою рабочую нагрузку, чтобы оценить, пригодится ли для нее архитектура Hyperscale. Она хорошо подходит для крупных баз данных OLTP, которые работают с относительно небольшим объемом горячих данных и нетребовательной рабочей нагрузкой. В то же время большие аналитические рабочие нагрузки обычно обрабатывают много данных, которые не помещаются в локальный кэш, а серверы страниц могут добавить накладные расходы, отчего снизится производительность.

Размер данных тоже имеет значение. Хотя это плохо задокументировано, каждый сервер страниц поддерживает до 1 Тбайт данных. Таким образом, эта архитектура не нужна для небольших баз данных, которые не будут должным образом масштабироваться на уровне хранилища. Но есть интересный сценарий использования: поскольку данные не сохраняются на вычислительных узлах, можно очень быстро подключать новые узлы и масштабировать рабочую нагрузку чтения. Так можно снизить затраты, если рабочая нагрузка в системе непостоянна.

В конечном итоге все продукты и уровни обслуживания Azure SQL работают на основе SQL Server. Поговорим о том, как устранять неполадки в этих системах.

Подходы к устранению неполадок

При устранении неполадок в управляемых облачных службах на основе SQL Server используется тот же набор метрик, что и для обычных экземпляров SQL Server, хотя собирать эти метрики приходится немного иначе. В обычном SQL Server устранение неполадок, как правило, начинается с анализа экосистемы и потребления ресурсов ОС. В управляемых облачных службах у вас нет доступа к метрикам ОС, но данные о потреблении ресурсов можно получить на портале Azure или через динамические административные представления.

На рис. 16.5 показан снимок экрана с метриками базы данных Azure SQL с портала Azure. (Я обрезал и увеличил нужные фрагменты, поэтому не удивляйтесь, что линии на графике разрываются.)

Аналогичную информацию можно получить из представления `sys.dm_db_resource_stats`[1], которое собирает данные каждые 15 секунд и хранит их примерно в течение часа. На рис. 16.6 показан пример его вывода.

В представлении `sys.resource_stats`[2] в базах данных Azure SQL или `sys.server_resource_stats`[3] в управляемых экземплярах Azure SQL можно посмотреть данные с меньшим разрешением, но за более долгое время (рис. 16.7).

---

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

[2] https://oreil.ly/luAvK

[3] https://oreil.ly/koZ6j

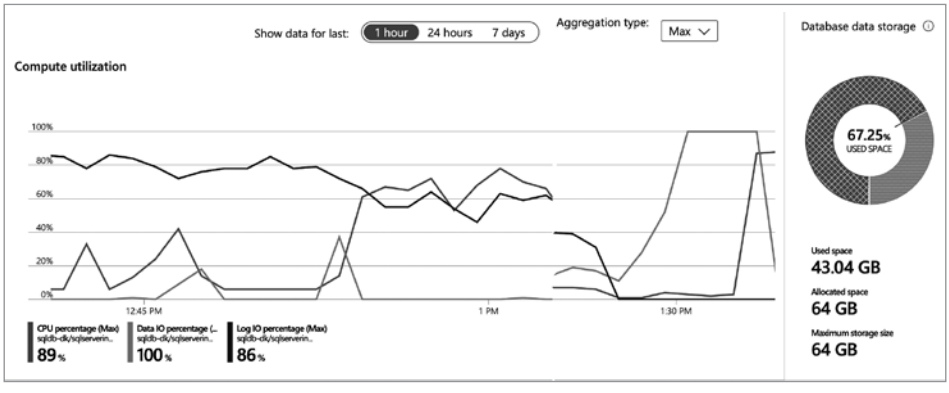


Рис. 16.5. Потребление ресурсов базы данных Azure SQL

	end_time	avg_cpu_percent	avg_data_io_percent	avg_log_write_percent	avg_memory_usage_percent	xtp_storage_percent	max_worker_percent
1	2021-11-24 18:38:09.600	76.89	0.00	0.00	32.22	0.00	1.25
2	2021-11-24 18:37:54.550	88.73	0.00	0.00	32.22	0.00	1.25
3	2021-11-24 18:37:39.553	88.59	0.00	0.00	32.22	0.00	1.25
4	2021-11-24 18:37:24.540	87.85	0.00	0.00	32.22	0.00	1.25
5	2021-11-24 18:37:09.517	88.66	0.00	0.00	32.22	0.00	1.25

max_session_percent	dtu_limit	avg_login_rate_percent	avg_instance_cpu_percent	avg_instance_memory_percent	cpu_limit	replica_role
0.04	NULL	NULL	76.05	81.78	4.00	0
0.04	NULL	NULL	76.05	81.78	4.00	0
0.04	NULL	NULL	75.37	81.78	4.00	0
0.04	NULL	NULL	75.37	81.78	4.00	0
0.04	NULL	NULL	75.37	81.78	4.00	0

Рис. 16.6. Вывод представления `sys.dm_db_resource_stats`

	start_time	end_time	database_name	sku	storage_in_megabytes
1	2021-11-24 18:28:53.9200000	2021-11-24 18:33:54.3166667	SQLServerInternals	GeneralPurpose	44072
2	2021-11-24 18:23:53.3233333	2021-11-24 18:28:53.9200000	SQLServerInternals	GeneralPurpose	65534
3	2021-11-24 18:18:52.8100000	2021-11-24 18:23:53.3233333	SQLServerInternals	GeneralPurpose	54863
4	2021-11-24 18:13:52.2333333	2021-11-24 18:18:52.8100000	SQLServerInternals	GeneralPurpose	46914
5	2021-11-24 18:08:51.7733333	2021-11-24 18:13:52.2333333	SQLServerInternals	GeneralPurpose	38389

avg_cpu_percent	avg_data_io_percent	avg_log_write_percent	max_worker_percent	max_session_percent	dtu_limit
32.19	65.06	0.00	1.25	0.04	NULL
2.65	19.16	7.64	2.75	0.04	NULL
7.10	8.99	32.46	2.75	0.04	NULL
8.95	6.46	46.39	2.75	0.04	NULL
18.92	11.53	37.61	2.75	0.04	NULL

xtp_storage_percent	avg_login_rate_percent	avg_instance_cpu_percent	avg_instance_memory_percent	cpu_limit	allocated_storage_in_megabytes
0.00	NULL	75.81	81.93	4.00	65536
0.00	NULL	17.85	81.92	4.00	65536
0.00	NULL	40.54	81.98	4.00	54864
0.00	NULL	41.29	81.94	4.00	48512
0.00	NULL	43.15	81.88	4.00	48512

Рис. 16.7. Вывод представления `sys.resource_stats`

Стабильно максимальный уровень потребления ресурсов в выходных данных указывает на то, что у базы данных или управляемого экземпляра не хватает пропускной способности для рабочей нагрузки, поэтому они лимитируются. В этом случае можно либо увеличить мощность экземпляра, чтобы повысить пропускную способность, либо отрегулировать систему, чтобы уменьшить нагрузку.

Как вы узнали в главе 6, в SQL Server 2017 и более поздних версиях можно включить автоматическое регулирование и разрешить SQL Server исправлять регрессивные планы, зависящие от параметров. Эта функция включена по умолчанию в базах данных Azure SQL, и ее можно включить в базах данных управляемых экземпляров. Можно также разрешить базе данных Azure SQL автоматически создавать индексы, чтобы бороться с неэффективными запросами. Система будет отслеживать эффективность индексов и сохранять или удалять их в зависимости от настроек (рис. 16.8).

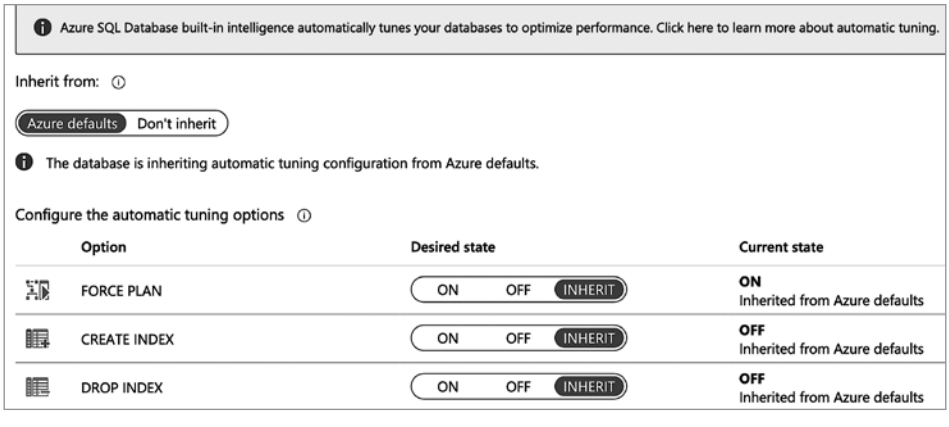


Рис. 16.8. Автоматическое регулирование в базе данных Azure SQL

Устранение неполадок на основе статистики ожидания работает с обеими управляемыми службами (в базе данных Azure SQL используйте представление `sys.dm_db_wait_stats`[1] вместо `sys.dm_os_wait_stats`). Вы увидите те же самые типы ожидания, что и в обычных экземплярах SQL Server. Другие динамические административные представления, рассмотренные в этой книге, тоже будут работать. Также можно использовать хранилище запросов, которое включено в базах данных Azure SQL по умолчанию и может быть включено в управляемых экземплярах.

Чтобы проанализировать задержки ввода/вывода, можно использовать представление `sys.dm_io_virtual_file_stats`, которое поможет понять, обеспечи-

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

вает ли хранилище Azure Blob достаточную пропускную способность на уровне обслуживания General Purpose. На уровне Hyperscale метрики, связанные с серверами страниц, добавляются к планам выполнения, к представлению `sys.dm_exec_requests` и представлениям статистики выполнения, а также в несколько расширенных событий.

Расширенные события можно использовать как в базах данных, так и в управляемых экземплярах Azure SQL. Имейте в виду, что для управления расширенными событиями базы данных Azure SQL используют несколько иной синтаксис T-SQL. Кроме того, из целевых объектов при этом поддерживаются только `ring_buffer`, `event_counter` и `event_file`. Управляемые экземпляры Azure SQL поддерживают все целевые объекты, как и обычный SQL Server. Используя целевой объект `event_file`, файлы нужно размещать в хранилище Azure Blob. Это может вызвать некоторую задержку, поэтому будьте осторожны, когда нужно регистрировать события, критичные для производительности.

В управляемых облачных службах нет доступа к `PerfMon`. Тем не менее метрики производительности SQL Server можно посмотреть с помощью представления `sys.dm_os_performance_counters`.

Наконец, в документации Microsoft[1] можно прочесть о некоторых других динамических административных представлениях, специфичных для Azure. В базах данных Azure SQL представление `sys.event_log` может заменить журнал ошибок SQL Server и содержит сведения о подключении и взаимных блокировках.

Amazon SQL Server RDS

Amazon Web Services предлагает несколько технологий в рамках своего семейства управляемых служб реляционных баз данных (RDS, Relational Database Services). Это семейство поддерживает несколько версий SQL Server, начиная с устаревшей версии 2012. Можно выбрать любую версию SQL Server, включая SQL Server Express.

Концептуально SQL Server RDS аналогичен управляемым экземплярам Azure SQL: вы работаете с виртуализированными экземплярами SQL Server, на которых может размещаться несколько баз данных. Хотя SQL Server RDS поддерживает большинство функций SQL Server, есть определенные ограничения: например, недоступны `FILESTREAM` или `UNSAFE CLR`.

SQL Server RDS допускает развертывание в нескольких зонах для высокой доступности. В этом режиме RDS создает другой экземпляр SQL Server в другой зоне доступности с помощью либо групп доступности, либо зеркального отображения базы данных с синхронной репликацией. Поскольку зоны доступности

[1] https://oreil.ly/ZG09I

полностью изолированы друг от друга, это может привести к дополнительной задержке фиксации, которая способна вызвать проблемы в системах с большим количеством транзакций.

В Enterprise Edition можно создавать доступные для чтения реплики в одном и том же регионе, чтобы масштабировать рабочую нагрузку чтения. Для поддержки этих реплик на внутреннем уровне в RDS применяется асинхронная репликация групп доступности. К сожалению, мультирегиональные реплики не поддерживаются, поэтому RDS не подходит, если система должна выдерживать сбои на уровне региона.

Некоторые административные задачи SQL Server RDS, в том числе перевод баз данных в оперативный и автономный режим, переименование баз данных и включение отслеживания измененных данных (CDC, Change Data Capture), требуют специальных хранимых процедур, которые доступны в базах данных `rdsadmin` и `msdb`. В частности, примечательна процедура `rds_read_error_log`, которая заменяет процедуру `xp_readerrorlog`. В документации AWS[1] приведен полный список задач и соответствующих хранимых процедур.

В консоли AWS есть несколько инструментов для быстрого мониторинга. Мы рассмотрим инструменты CloudWatch и Performance Insights. Еще один инструмент, Enhanced Monitoring, собирает данные о потреблении памяти и хранилища в виртуальном экземпляре. Это важный источник информации, но его сведения довольно просты, поэтому в этом разделе основное внимание уделяется первым двум инструментам.

CloudWatch

CloudWatch предоставляет данные об общем потреблении ресурсов на уровне гипервизора, включая нагрузку ЦП, пропускную способность ввода/вывода и сети, а также использование хранилища (рис. 16.9). Высокий уровень потребления ресурсов указывает на то, что экземпляр перегружен и его следует расширить или отрегулировать.

Обратите внимание на метрики ввода/вывода: производительность хранилища часто становится главным узким местом в облаке. Это особенно важно, если вы оснащаете экземпляр RDS дисками общего назначения (General Purpose). У этого типа хранилища предусмотрена резервная мощность, которая позволяет компенсировать всплески рабочей нагрузки ввода/вывода. Однако после исчерпания резервной мощности производительность диска сильно лимитируется, из-за чего возникают периодические проблемы с производительностью в SQL Server. Подумайте о том, чтобы перейти на подготовленные диски IOPS, которые стоят дороже, но обеспечивают предсказуемую производительность.

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

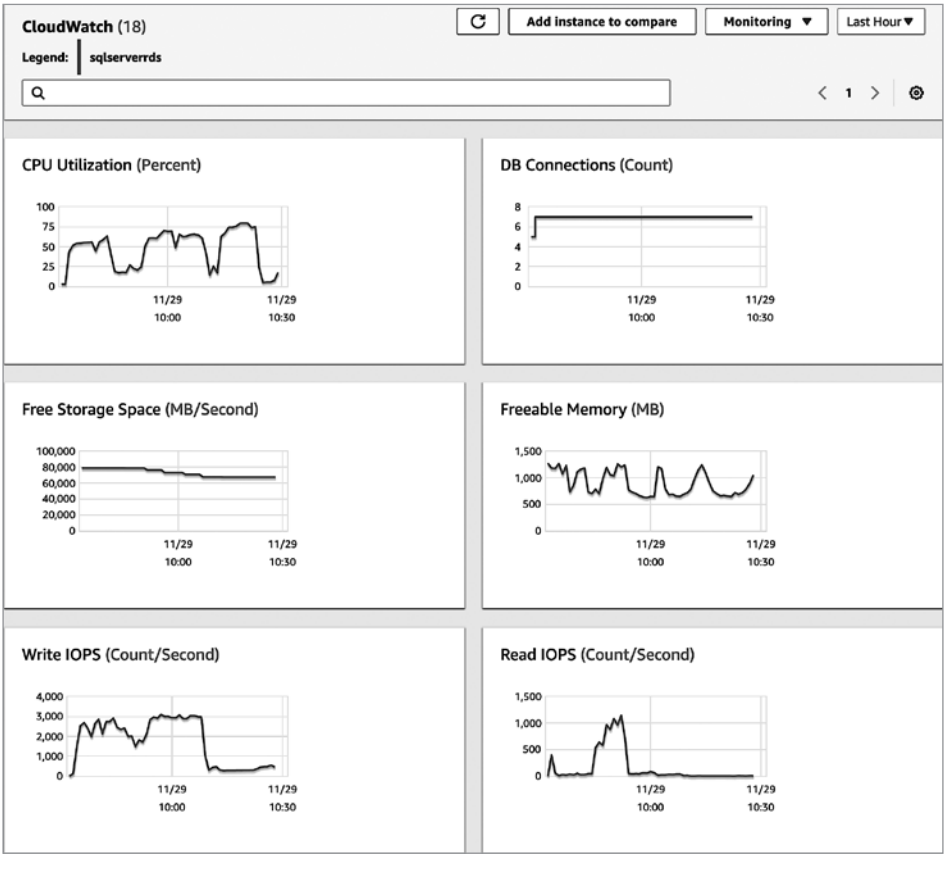


Рис. 16.9. Метрики CloudWatch

Performance Insights

Performance Insights — это очень основательный инструмент мониторинга, предоставляющий информацию по нескольким направлениям. Прежде всего он отображает некоторые счетчики производительности ОС и SQL Server (рис. 16.10), подобно утилите `PerfMon`.

На втором графике представлена подробная информация о нагрузке базы данных в зависимости от времени. Данные можно группировать по типу ожидания, инструкциям SQL, а также по пользователю и узлу, как показано на рис. 16.11.

Третий график — еще одна проекция нагрузки базы данных — отображает самые ресурсоемкие запросы к базе (рис. 16.12), самые популярные ожидания и самых активных пользователей.

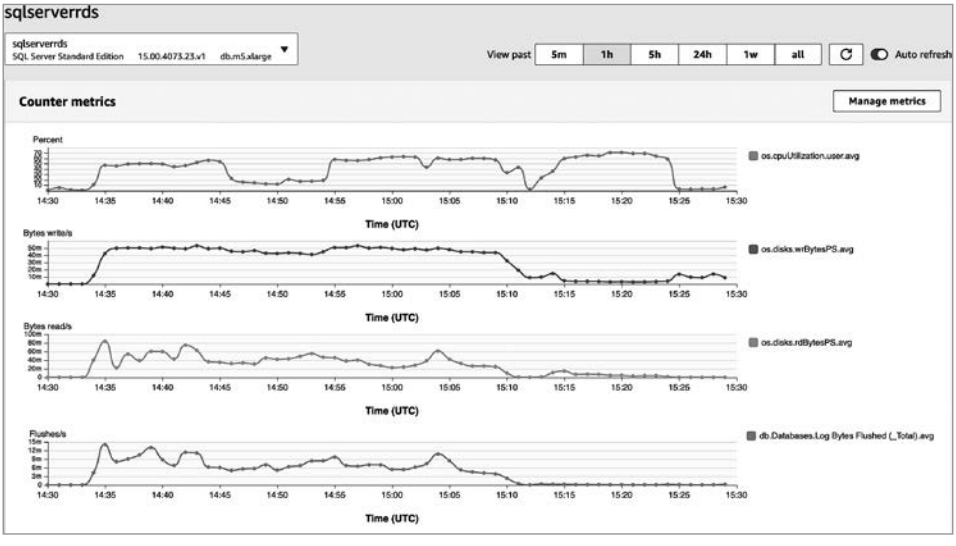


Рис. 16.10. Performance Insights: метрики производительности

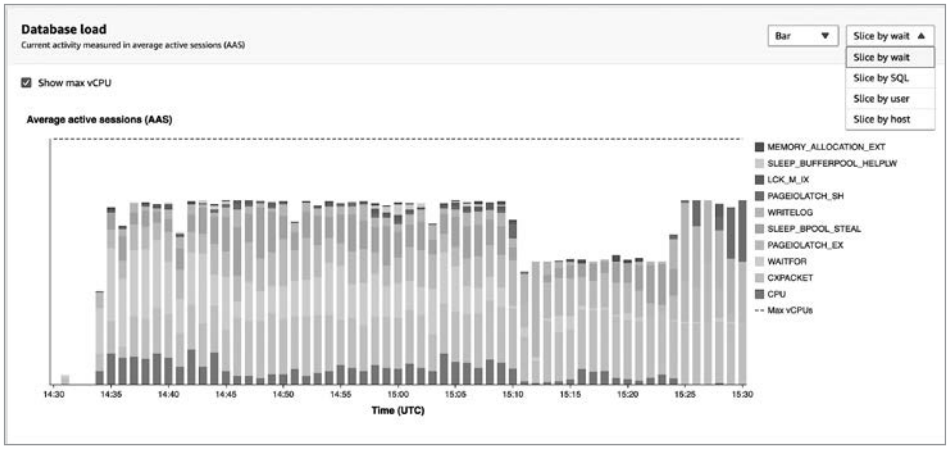


Рис. 16.11. Performance Insights: информация о нагрузке базы данных

В целом Performance Insights по своим возможностям находится на одном уровне с утилитами базового мониторинга SQL Server. Этот инструмент полезен для быстрого устранения неполадок, когда нужно оценить общее состояние сервера и выявить наименее производительные запросы.

В RDS прекрасно работают стандартные методы устранения неполадок SQL Server. Можно запрашивать динамические административные представления,

использовать хранилище запросов и применять расширенные события точно так же, как в обычных экземплярах SQL Server. В качестве целевого объекта расширенных событий можно указывать `event_file`; при этом данные будут сохраняться в папку *D:\RDSDATA\LOG*.

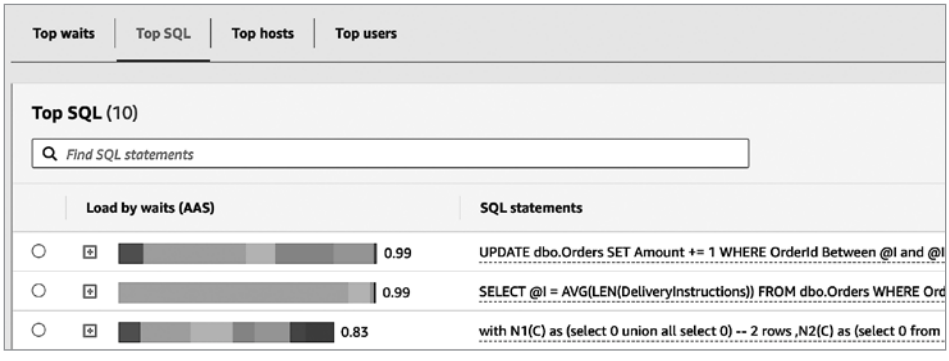


Рис. 16.12. Performance Insights: самые ресурсоемкие запросы

Google Cloud SQL

Аналогично AWS и Microsoft Azure, Google тоже предлагает службы управляемых баз данных на своей платформе. Google SQL поддерживает SQL Server примерно так же, как Amazon RDS и управляемые экземпляры Azure SQL, предоставляя виртуализированный экземпляр SQL Server, на котором можно создавать несколько баз данных.

Google Cloud Platform поддерживает высокую доступность за счет резервного экземпляра SQL Server в другой зоне доступности и репликации на уровне хранилища. При этом не возникает накладных расходов на репликацию групп доступности AlwaysOn, но хранилище реплицируется синхронно, что может привести к задержке фиксации, как и у других поставщиков. В Enterprise Edition также можно подготовить доступные для чтения реплики, в которых на внутреннем уровне используются группы доступности AlwaysOn.

Google официально начал поддерживать SQL Server в своем семействе облачных служб SQL в 2021 году, и его набор инструментов для мониторинга и устранения неполадок был еще недостаточно зрелым по состоянию на декабрь 2021 года, когда писались эти строки. Инструменты мониторинга Google Cloud Platform довольно просты и ограничены базовыми метриками потребления ресурсов (см. рис. 16.13). Тем не менее я надеюсь, что со временем мониторинг будет развиваться и «дозреет» до уровня других технологий баз данных, предоставляемых этой платформой.

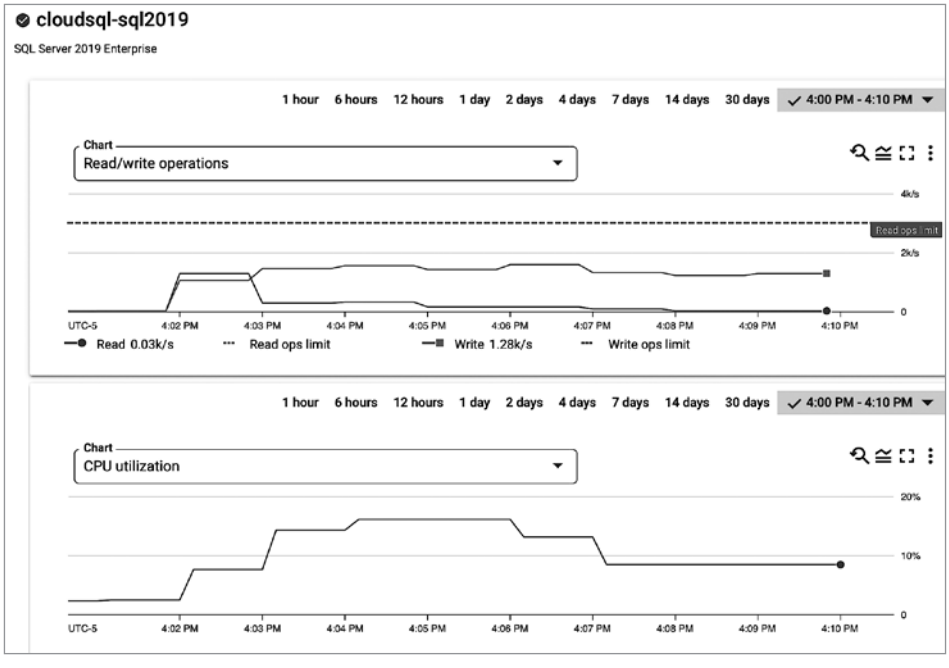


Рис. 16.13. Графики потребления ресурсов Google Cloud SQL

В Google Cloud SQL можно использовать сторонние инструменты мониторинга, а можно устранять неполадки с помощью собственных функций SQL Server, таких как динамические административные представления и хранилище запросов. Однако у этой службы есть критическое ограничение: у учетной записи администратора, предоставленной платформой, нет прав на создание расширенных событий или трассировок SQL.

Резюме

Все ведущие поставщики облачных услуг позволяют запускать SQL Server на облачных виртуальных машинах и предлагают управляемые службы баз данных, которые обеспечивают высокую доступность, резервное копирование и обслуживание SQL Server и ОС. На внутреннем уровне все эти системы используют ядро SQL Server и позволяют применять те же методы настройки и устранения неполадок, что и на обычных локальных экземплярах SQL Server.

Каждая служба предоставляет доступ к облачной консоли, чтобы отслеживать потребление ресурсов виртуальной машины SQL Server или управляемых экземпляров базы данных. Интенсивное потребление ресурсов приведет к лимити-

рованию производительности, которое можно преодолеть, если отрегулировать или нарастить систему.

Обращайте особое внимание на производительность подсистемы хранения. Для рабочих нагрузок баз данных используйте хранилище с гарантированной производительностью. Если вы запускаете SQL Server в облачной виртуальной машине, попробуйте создать в ОС массивы RAID-0, что может обеспечить лучшее соотношение цены и производительности, чем отдельные диски.

Независимо от поставщика убедитесь, что приложения устойчивы и способны обрабатывать случайные ошибки. Сбои обязательно будут происходить, и приложения должны уметь восстанавливаться после них.

В системах, распределенных по нескольким регионам, учитывайте задержку межрегиональной связи. Не используйте синхронную репликацию между регионами. Запускайте пакетные операции T-SQL, чтобы уменьшить количество межрегиональных вызовов, и не используйте доступные для чтения вторичные реплики, если вам нужны актуальные данные.

Чек-лист устранения неполадок

- ☐ Проверить, удовлетворяет ли настройка высокой доступности экземпляра базы данных требуемому соглашению об уровне обслуживания (SLA) в случае аварии на уровне зоны доступности или региона.
- ☐ Проверить конфигурацию экземпляра и потребление ресурсов, уделяя особое внимание настройке и производительности хранилища.
- ☐ Проверить стратегии обслуживания индексов и статистики даже при использовании управляемых служб.
- ☐ Выполнить общие тесты работоспособности SQL Server так же, как в случае с обычным локальным экземпляром.

Еще раз благодарю вас за то, что прочли эту книгу! Надеюсь, она вам понравилась. SQL Server — отличная технология, а устранение неполадок и тонкая настройка — один из самых увлекательных и полезных аспектов нашей работы. Для меня было удовольствием и честью писать эту книгу для вас.

ПРИЛОЖЕНИЕ

Типы ожиданий

В этом приложении перечислены наиболее распространенные типы ожиданий, которые встретятся вам при устранении неполадок. Я опишу условия, при которых могут произойти ожидания, а также дам общие советы по устранению неполадок.

Это приложение не заменяет документацию Microsoft[1] или библиотеку типов ожиданий SQLSkills[2]. Но думаю, что этот справочный материал хорошо дополняет книгу.

ASYNC_IO_COMPLETION

Ожидание `ASYNC_IO_COMPLETION` происходит во время асинхронных операций ввода/вывода, не связанных с буферным пулом. Вот его распространенные причины:

- Обычные контрольные точки.
- Внутренние контрольные точки, которые возникают, когда вы запускаете резервное копирование базы данных или `DBCC CHECKDB`.
- Чтение страниц глобальной карты распределения (GAM) из файлов данных.
- Чтение страниц данных из базы данных во время ее резервного копирования (к сожалению, информация о среднем времени ожидания при этом искажается, что затрудняет анализ).

Заметный процент ожиданий `ASYNC_IO_COMPLETION` может быть признаком перегруженной подсистемы ввода/вывода, особенно если он проявляется вместе с другими ожиданиями, связанными с вводом/выводом. Устраните неисправности системы ввода/вывода (см. ожидания `PAGEIOLATCH` далее в этом приложении).

Дополнительная информация приведена в главе 3.

---

[1] https://oreil.ly/6oto3

[2] https://oreil.ly/Wt5Mx

ASYNC_NETWORK_IO

Ожидание ASYNC_NETWORK_IO возникает, когда SQL Server ждет, пока клиент примет данные. Три самые распространенные причины этого ожидания таковы:

- Медленная сеть — например, когда клиенты подключены к серверу баз данных, размещенному в другом дата-центре или в облаке.
- Клиентские приложения, работающие на недостаточно мощных виртуальных машинах или на перегруженных серверах.
- Неправильно спроектированное приложение, когда клиент обрабатывает данные построчно.

Это ожидание не всегда означает, что проблема на стороне сервера. Тем не менее неэффективное взаимодействие с клиентом потребляет рабочие потоки и другие ресурсы SQL Server. Когда наблюдается значительный процент таких ожиданий, стоит выполнить устранение неполадок.

Этапы устранения неполадок:

- Проверьте производительность сети на сервере, просматривая счетчики производительности сети и другие доступные ключевые индикаторы производительности (KPI).
- Найдите клиентские приложения, которые генерируют ожидания, с помощью представлений sys.dm_os_waiting_tasks, sys.dm_exec_requests, sys.dm_exec_sessions и sys.dm_exec_connections (можно использовать листинг 2.3). Для краткосрочного профилирования используйте расширенное событие wait_completed с захватом действий sqlserver.client_host_name и sqlserver.client_app_name.
- Проанализируйте код приложения и производительность серверов приложений.

Дополнительная информация приведена в главах 13 и 16.

BACKUPBUFFER

См. тип ожидания BACKUPIO.

BACKUPIO

Ожидания BACKUPIO и BACKUPBUFFER указывают на недостаточную пропускную способность резервного копирования и/или восстановления. Они могут возникать из-за медленной или перегруженной сетевой и/или дисковой подсистем. В последнем случае они часто появляются вместе с другими ожиданиями, связанными с вводом/выводом.

Если наблюдаются ожидания, связанные с резервным копированием, устраняйте неполадки производительности процесса резервного копирования. Проанализируйте производительность диска и сетевой подсистемы. Попробуйте внедрить сжатие резервных копий и/или резервное копирование с чередованием, а также отрегулировать параметры резервного копирования.

Дополнительная информация приведена в главе 13.

BTREE_INSERT_FLOW_CONTROL

Ожидание `BTREE_INSERT_FLOW_CONTROL` указывает на существование индексов с постоянно увеличивающимися ключами, которые создают «горячие точки». Это ожидание заменяет некоторые ожидания `PAGELATCH`, когда вы включаете параметр индекса `OPTIMIZE_FOR_SEQUENTIAL_KEY`.

Ознакомьтесь с примечаниями по устранению неполадок для ожиданий `PAGELATCH`, а также с дополнительной информацией в главе 10.

CXCONSUMER

Ожидания `CXPACKET`, `CXCONSUMER` и `EXCHANGE` возникают во время выполнения запросов с параллельными планами выполнения. См. тип ожидания `CXPACKET` и подробные объяснения в главе 6.

CXPACKET

Ожидания `CXPACKET`, `CXCONSUMER` и `EXCHANGE` возникают во время выполнения запросов с параллельными планами выполнения. Эти ожидания совершенно нормальны и должны присутствовать, особенно в системах с рабочими нагрузками хранения данных и/или отчетов. Однако в системах OLTP чрезмерное количество ожиданий параллелизма может создавать неприятности.

Ожидания параллелизма сами по себе не являются проблемой: они всего лишь признак ресурсоемких запросов, которые используют планы параллельного выполнения. В системах OLTP такие запросы нужно обнаружить (см. главу 4) и оптимизировать (см. главу 5). Настройте параметры параллелизма. Не устанавливайте параметр `MAXDOP=1`, потому что это просто замаскирует проблему.

Дополнительная информация приведена в главе 6.

DIRTY_PAGE_TABLE_LOCK

Ожидания `DIRTY_PAGE_TABLE_LOCK`, `DPT_ENTRY_LOCK`, `PARALLEL_REDO_FLOW_CONTROL` и `PARALLEL_REDO_TRAN_TURN` могут возникать на доступных для чтения вторичных узлах в группах доступности AlwaysOn.

Большое количество таких ожиданий может сигнализировать о проблемах с процессом параллельного повтора. Когда это происходит, ожидания наблюдаются в представлениях `sys.dm_os_waiting_tasks` и `sys.dm_exec_requests` и сопровождаются повышенной нагрузкой ЦП и ростом очереди повтора. Единственный способ борьбы с этой проблемой, известный мне на данный момент, — применить исправления SQL Server и/или отключить параллельный повтор с помощью флага трассировки `T3459`.

Дополнительная информация приведена в главе 12.

DPT_ENTRY_LOCK

См. тип ожидания `DIRTY_PAGE_TABLE_LOCK`.

EXCHANGE

Ожидания `CXPACKET`, `CXCONSUMER` и `EXCHANGE` возникают во время выполнения запросов с параллельными планами выполнения. См. тип ожидания `CXPACKET` и подробные объяснения в главе 6.

HADR_GROUP_COMMIT

Ожидание `HADR_GROUP_COMMIT` происходит, когда первичный узел группы доступности AlwaysOn пытается оптимизировать производительность репликации, группируя вместе несколько фиксационных записей журнала перед тем, как отправить их на вторичные узлы.

Это ожидание редко создает проблемы. Если высокий процент этого типа ожидания начинает влиять на пропускную способность системных транзакций, нужно устранить неполадки с производительностью репликации AlwaysOn (см. ожидание `HADR_SYNC_COMMIT`). Можно также отключить групповую фиксацию с помощью флага трассировки `T9546`.

Дополнительная информация приведена в главе 12.

HADR_SYNC_COMMIT

Ожидание `HADR_SYNC_COMMIT` происходит, когда первичный узел группы доступности AlwaysOn ждет, пока синхронные вторичные узлы закрепят записи журнала транзакций. Эти ожидания всегда присутствуют в группах доступности, использующих синхронную репликацию.

Проанализируйте среднюю продолжительность этих ожиданий. Она должна быть как можно меньше: не более нескольких миллисекунд, если синхронные реплики размещены в том же дата-центре, что и первичный узел. Если наблюдаются бóльшие значения, проанализируйте производительность репликации,

обращая внимание на задержку и пропускную способность сети, производительность диска и общую нагрузку на синхронные реплики.

Транзакции остаются активными, и на них удерживаются все блокировки до тех пор, пока первичный узел не получит подтверждение того, что запись журнала транзакций `COMMIT` закреплена на синхронных вторичных узлах. Таким образом, большая продолжительность ожидания `HADR_SYNC_COMMIT` может усугублять блокирование в системе.

Дополнительная информация приведена в главе 12.

HTBUILD

Ожидания `HTBUILD`, `HTDELETE`, `HTMEMO`, `HTREINIT` и `HTREPARTITION` возникают во время управления внутренними хеш-таблицами в пакетном режиме выполнения. В средах с индексами columnstore эти ожидания могут говорить о плохом обслуживании индексов columnstore, когда образуются большие разностные хранилища и/или группы строк неравномерного размера. Проанализируйте состояние групп строк с помощью представления `sys.column_store_row_group` и при необходимости перестройте индексы.

Дополнительная информация приведена в главе 13.

HTDELETE, HTMEMO, HTREINIT и HTREPARTITION

См. тип ожидания `HTBUILD`.

IO_COMPLETION

Тип ожидания `IO_COMPLETION` возникает во время синхронного чтения и записи в файлы данных, а также во время некоторых операций чтения в журнале транзакций. Вот несколько примеров:

- Чтение страниц карты распределения из базы данных.
- Чтение журнала транзакций во время восстановления базы данных.
- Запись данных в `tempdb` во время переноса при сортировке.

Заметный процент ожиданий `IO_COMPLETION` может указывать на перегруженную подсистему ввода/вывода, особенно если при этом наблюдаются и другие ожидания, связанные с вводом/выводом. Устраните неисправности системы ввода/вывода (см. ожидание `PAGEIOLATCH`). Обратите особое внимание на задержку и пропускную способность `tempdb`. Плохая производительность `tempdb` — одна из наиболее частых причин этого ожидания.

Дополнительная информация приведена в главах 3 и 9.

LATCH_*

Ожидания, имена которых начинаются с LATCH_, вызываются кратковременными блокировками, не связанными с буферным пулом. SQL Server генерирует разные ожидания LATCH в зависимости от типа кратковременных блокировок (совмещаемые, монопольные и т. д.).

Чтобы получить статистику и подробности о кратковременных блокировках, а также анализировать узкие места, можно использовать представление sys.dm_os_latch_stats.

Дополнительная информация приведена в главе 10.

LCK_M_*

Ожидания, имена которых начинаются с LCK_M_, происходят во время блокирования. Каждому типу блокировки в SQL Server соответствует тип ожидания.

Проанализируйте, когда используется тот или иной тип блокировки, и определите и устраните основную причину блокирования.

В этом приложении подробно описаны ожидания LCK_M_I*, LCK_M_R*, LCK_M_S, LCK_M_SCH_M, LCK_M_SCH_S, LCK_M_U и LCK_M_X.

Дополнительная информация приведена в главе 8.

LCK_M_I*

SQL Server устанавливает интентные блокировки на уровне объекта (таблицы) и страницы. На уровне таблицы ожидания LCK_M_I* обычно происходят в двух случаях:

- Несовместимость с блокировками модификации схемы. В этом случае обычно также наблюдаются ожидания блокировки схемы (LCK_M_SCH_S и LCK_M_SCH_M). Подробнее см. ожидание LCK_M_SCH_M.
- Полная несовместимая блокировка на уровне таблицы, удерживаемая другим сеансом. Обычно это происходит из-за укрупнения блокировки или из-за кода, использующего указания TABLOCK и TABLOCKX. Для устранения неполадок с укрупнением блокировки можно использовать Blocking Monitoring Framework и представление sys.dm_index_operational_stats (см. главу 14).

Ожидания интентных блокировок, возникающие из-за блокирования на уровне страницы, обычно говорят о неоптимизированных запросах и просмотрах. Анализируйте отдельные случаи блокирования, чтобы найти их основную причину.

Дополнительная информация приведена в главе 8.

LCK_M_R*

Типы ожидания LCK_M_R* указывают на ожидание блокировки диапазона. SQL Server устанавливает такие блокировки на уровне изоляции SERIALIZABLE. Еще одна возможная причина ожиданий — некластеризованные индексы, для которых установлен параметр IGNORE_DUP_KEY=ON.

Увидев эти ожидания, определите и устраните основную причину блокирования. Избегайте параметра IGNORE_DUP_KEY=ON в некластеризованных индексах. Не используйте уровень изоляции SERIALIZABLE без крайней необходимости.

Дополнительная информация приведена в главе 8.

LCK_M_S

Тип ожидания LCK_M_S указывает на ожидание совмещаемых блокировок (S). Этот тип блокировки устанавливается запросами SELECT на уровнях изоляции READ COMMITTED, REPEATABLE READ и SERIALIZABLE. Неоптимизированные запросы SELECT — самый распространенный источник этих ожиданий.

Если наблюдается большое количество таких ожиданий, сосредоточьтесь на оптимизации запросов (глава 5). Когда запросы выполняются на уровне изоляции READ COMMITTED, попробуйте включить параметр базы данных READ_COMMITTED_SNAPSHOT, чтобы устранить блокирование между операциями чтения и записи. (Правда, это лишь скроет проблему с неоптимизированными запросами, а не решит ее.)

Дополнительная информация приведена в главе 8.

LCK_M_SCH_M

Тип ожидания LCK_M_SCH_M указывает на ожидание блокировок модификации схемы (Sch-M), когда сеансы не могут получить монопольные блокировки для объектов.

Если встречаются эти ожидания, проанализируйте свою стратегию развертывания для изменений базы данных. Также обратите внимание на стратегии обслуживания индексов и разделов; эти операции устанавливают блокировки таблиц. Во время перестроения индекса и переключения разделов попробуйте использовать низкоприоритетные блокировки, если эта функция поддерживается. Наконец, проанализируйте отдельные случаи блокирования, чтобы определить основную причину проблемы.

Посмотрите, как часто возникают эти ожидания. Они могут быть вызваны однократными событиями и ошибками во время развертывания.

Дополнительная информация приведена в главе 8.

LCK_M_SCH_S

Тип ожидания LCK_M_SCH_S указывает на ожидание блокировки стабильности схемы (Sch-S). Этот тип блокирования возникает во время модификации схемы, когда другие сеансы удерживают блокировки модификации схемы (Sch-M) на объектах.

См. стратегию устранения неполадок типа ожидания LCK_M_SCH_M и подробные объяснения в главе 8.

LCK_M_U

Тип ожидания LCK_M_U указывает на ожидание блокировок обновления (U). SQL Server устанавливает их во время просмотра обновлений, обычно вызванного неоптимизированными запросами записи (UPDATE, DELETE, MERGE). Во многих случаях это сопровождается типами ожидания PAGEIOLATCH* и CXPACKET.

Если наблюдается много таких ожиданий, сосредоточьтесь на оптимизации запросов (глава 5).

Дополнительная информация приведена в главе 8.

LCK_M_X

Тип ожидания LCK_M_X указывает на ожидание монопольных блокировок (X). Распространенные причины таких ожиданий — искусственные точки сериализации (таблицы счетчиков), злоупотребление уровнями изоляции REPEATABLE READ и SERIALIZABLE, неэффективное управление транзакциями и длительные транзакции, а также указания блокировки на уровне таблицы, такие как (TABLOCKX).

Если наблюдается много таких ожиданий, проанализируйте отдельные случаи блокирования, чтобы определить основную причину проблемы.

Дополнительная информация приведена в главе 8.

LOGBUFFER

Ожидание LOGBUFFER возникает, когда SQL Server ожидает доступный буфер журнала, чтобы внести записи журнала. Обычно это ожидание идет в паре с ожиданием WRITELOG и указывает на недостаточную пропускную способность журнала транзакций. См. ожидание WRITELOG.

Дополнительная информация приведена в главах 3 и 11.

OLEDB

Ожидание OLEDB происходит, когда SQL Server ожидает данных от поставщика OLEDB. Чаще всего оно наблюдается в таких случаях:

- Вызовы на связанные серверы.
- Выполнение некоторых пакетов SQL Server Integration Services (SSIS).
- Операции во время выполнения `DBCC CHECKDB`.
- Запросы к динамическим административным представлениям.

Эти ожидания не всегда являются проблемой, но нужно понять, чем они вызваны и насколько сильно влияют на систему.

Дополнительная информация приведена в главе 13.

PAGEIOLATCH*

Ожидания `PAGEIOLATCH*` возникают, когда SQL Server считывает страницы данных с диска. Эти ожидания очень распространены и наблюдаются в большинстве систем. Существует шесть типов ожидания `PAGEIOLATCH`, которые относятся к разным видам операций со страницами буферного пула. Анализируйте ожидания всех типов `PAGEIOLATCH` совместно, чтобы оценить их совокупное влияние.

Чрезмерное количество ожиданий `PAGEIOLATCH` показывает, что SQL Server постоянно считывает данные с диска. Обычно это происходит в двух случаях:

- недостаточное аппаратное обеспечение SQL Server, когда активные данные не помещаются в памяти;
- неоптимизированные запросы, которые просматривают ненужные данные, сбрасывая содержимое буферного пула на диск.

Можно выполнить перекрестную проверку данных, изучив счетчик производительности `Page Life Expectancy`, который показывает, как долго страницы данных остаются в буферном пуле.

Значительный процент ожиданий `PAGEIOLATCH` всегда требует устранения неполадок. Эти ожидания не обязательно создают проблемы для клиентов, особенно при использовании флеш-накопителей с малой задержкой, однако рост данных может привести к тому, что дисковая подсистема перестанет справляться, и тогда проблема очень быстро затронет всю систему.

Как устранить проблему:

- Проверьте производительность и задержку дисковой подсистемы с помощью представления `sys.dm_io_virtual_file_stats` (листинг 3.1).
- Проверьте, не вызвана ли высокая задержка всплесками активности ввода/вывода, проанализировав метрики производительности SQL Server и ОС. При необходимости отрегулируйте процесс контрольной точки.

- Проанализируйте весь стек ввода/вывода и найдите узкие места.
- Определите и отрегулируйте запросы с интенсивным вводом/выводом (главы 4 и 5).
- Проанализируйте метрики индекса (глава 14), если нужно обнаружить индексы, которые порождают бóльшую часть этих ожиданий, и/или определите самых активных потребителей буферного пула.

Дополнительная информация приведена в главах 3 и 15.

PAGELATCH

Ожидания `PAGELATCH` указывают на блокировки, связанные с буферным пулом, которые происходят, когда потокам необходимо одновременно получить доступ к данным (или страницам карт распределения) или модифицировать их. Две основные причины таких ожиданий — состязания на системных страницах `tempdb` и «горячие точки» в постоянно растущих индексах.

Когда наблюдаются эти ожидания, разберитесь, что их вызывает. Чтобы проверить, связаны ли они с `tempdb`, проанализируйте столбец `wait_resource` в представлении `sys.dm_os_waiting_tasks`. Также можно перехватить расширенное событие `sqlserver.latch_suspend_end`. Устраняя проблемы с кратковременными блокировками в `tempdb`, убедитесь, что `tempdb` настроена правильно, уменьшите нагрузку на нее и попробуйте включить метаданные `tempdb`, оптимизированные для памяти, если такая функция поддерживается.

Индексы с «горячими точками» можно найти с помощью функции `sys.dm_db_index_operational_stats`. В SQL Server 2019 и более поздних версиях можно уменьшить состязания, включив настройку индекса `OPTIMIZE_FOR_SEQUENTIAL_KEY`. Также можно провести рефакторинг схемы, внедрить секционирование хешей или использовать OLTP в памяти.

Дополнительная информация приведена в главах 9 и 10.

PARALLEL_REDO_FLOW_CONTROL

См. тип ожидания `DIRTY_PAGE_TABLE_LOCK`.

PARALLEL_REDO_TRAN_TURN

См. тип ожидания `DIRTY_PAGE_TABLE_LOCK`.

PREEMPTIVE_OS_ACCEPTSCURITYCONTEXT

См. типы ожидания `PREEMPTIVE_OS_AUTH*`.

PREEMPTIVE_OS_AUTH*

Ожидания с именами, начинающимися с PREEMPTIVE_OS_AUTH, PREEMPTIVE_OS_LOOKUPACCOUNTSID и PREEMPTIVE_OS_ACCEPTSCURITYCONTEXT, происходят во время аутентификации пользователей.

Если наблюдается значительный процент таких ожиданий, проверьте производительность контроллеров Active Directory. Убедитесь, что SQL Server не выполняет аутентификацию на контроллерах Active Directory в удаленных дата-центрах.

Также проверьте, не используется ли в коде контекст EXECUTE AS OWNER или EXECUTE AS USER, который инициирует вызовы аутентификации.

Дополнительная информация приведена в главе 13.

PREEMPTIVE_OS_LOOKUPACCOUNTSID

См. типы ожидания PREEMPTIVE_OS_AUTH*.

PREEMPTIVE_OS_WRITEFILE

Ожидание PREEMPTIVE_OS_WRITEFILE может быть признаком узкого места во время синхронной записи в файлы. Обнаружив эти ожидания, проверьте, не выполняет ли сервер несколько трассировок или аудитов SQL, используя файлы как целевые объекты для сохранения данных. Также проверьте наличие моментальных снимков базы данных — это другая причина таких ожиданий.

Дополнительная информация приведена в главе 13.

PREEMPTIVE_OS_WRITEFILEGATHER

Ожидание PREEMPTIVE_OS_WRITEFILEGATHER происходит во время процесса инициализации нулями. Когда в системе возникает это ожидание, проверьте и включите мгновенную инициализацию файла, предоставив учетной записи SQL Server разрешение *Perform Volume Management Tasks* (SE_MANAGE_VOLUME_NAME).

Изучите параметры автоувеличения журнала транзакций и убедитесь, что нет процессов, которые регулярно сокращают журнал.

Дополнительная информация приведена в главе 13.

QDS*

Семейство типов ожидания, имена которых начинаются с QDS, связано с хранилищем запросов. Они могут быть признаком накладных расходов из-за сбора данных хранилища запросов. Ожидания QDS_PERSIST_TASK_MAIN_LOOP_SLEEP и QDS_ASYNC_QUEUE можно игнорировать — они безвредные.

Если наблюдаются другие ожидания QDS, проверьте настройки хранилища запросов. Не используйте режим сбора данных QUERY_CAPTURE_MODE=ALL. Уменьшите размер хранилища запросов, если оно слишком велико. Также включите флаги трассировки T7745 и T7752.

Дополнительная информация приведена в главе 4.

RESOURCE_SEMAPHORE

Ожидание RESOURCE_SEMAPHORE происходит, когда запросы ожидают предоставления памяти. Этот тип ожидания всегда стоит изучать, если он становится заметен в системе.

Обнаружив это ожидание, проанализируйте использование памяти в SQL Server. Проверьте, как клерки используют память и нет ли признаков внешней и внутренней нехватки памяти. В виртуализированных средах проверьте поведение драйвера накачки. Убедитесь, что сервер адекватно оснащен, а SQL Server правильно настроен под рабочую нагрузку.

Ожидания RESOURCE_SEMAPHORE также могут вызываться отдельными запросами. Проанализируйте объемы предоставленной памяти с помощью представления sys.dm_exec_query_memory_grants и при необходимости оптимизируйте запросы.

Дополнительная информация приведена в главе 7.

RESOURCE_SEMAPHORE_QUERY_COMPILE

Ожидание RESOURCE_SEMAPHORE_QUERY_COMPILE происходит, когда SQL Server не хватает памяти для компиляции запросов. Как и в случае с RESOURCE_SEMAPHORE, необходимо изучить проблему.

Проверьте, не выполняет ли SQL Server слишком много компиляций. Для этого просмотрите счетчики производительности SQL Compilations/sec и SQL Recompilations/sec. Сократите количество компиляций за счет параметризации запросов (глава 6). Выполните общее устранение неполадок, связанных с памятью, как описано в статье об ожидании RESOURCE_SEMAPHORE.

Я также замечал, что ожидание RESOURCE_SEMAPHORE_QUERY_COMPILE возникает, когда очень активная таблица удерживает блокировку (Sch-M) во время длительного процесса автономного перестроения индекса. Одновременно с этим SQL Server пытается перекомпилировать большое количество запросов, обращающихся к этой таблице. Компиляции блокируются и в конце концов потребляют огромное количество памяти, из-за чего остальным заявкам на компиляцию приходится ждать с этим типом ожидания.

Дополнительная информация приведена в главе 7.

THREADPOOL

Ожидание THREADPOOL возникает, когда у SQL Server нет доступных рабочих процессов для обслуживания запросов пользователей. Это опасное ожидание, которое необходимо исследовать.

Чаще всего оно возникает в следующих случаях:

- неправильная настройка параметра *max worker threads*;
- недостаточный объем памяти SQL Server (проверьте конфигурацию ОС и SQL Server);
- длинные цепочки блокирования;
- чрезмерная нехватка памяти;
- большое количество подключенных клиентов;
- нагрузка с чрезмерным количеством запросов с параллельными планами выполнения.

В облачных средах эти ожидания могут указывать на то, что база данных работает на недостаточно оснащенном узле, который не справляется с нагрузкой.

Дополнительная информация приведена в главах 13 и 16.

WRITE_COMPLETION

Ожидание WRITE_COMPLETION возникает во время операций синхронной записи в файлы данных и журналов. По моему опыту, чаще всего это происходит при создании снимков базы данных.

Когда в системе возникает это ожидание, проверьте, не создаются ли снимки базы. Помните, что некоторые внутренние процессы, такие как DBCC CHECKDB, создают внутренние снимки базы.

Когда ожидание WRITE_COMPLETION наблюдается вместе с другими ожиданиями, связанными с вводом/выводом, следует устранить неполадки системы ввода/вывода (см. ожидания PAGEIOLATCH).

Дополнительная информация приведена в главе 3.

WRITELOG

Ожидание WRITELOG возникает, когда SQL Server вносит записи журнала в журнал транзакций. Это ожидание нормально в любой системе, но его значительный процент и/или большое среднее время ожидания могут быть признаком узкого места в журнале транзакций. Ожидание WRITELOG часто наблюдается в паре с ожиданием LOGBUFFER, а оно тоже является признаком узкого места.

Чтобы устранить проблему, проанализируйте среднее время ожидания и задержку записи журнала транзакций с помощью представления `sys.dm_io_virtual_file_stats` (листинг 3.1). Высокие значения этих метрик могут повлиять на пропускную способность системы. Устраните неполадки с производительностью журнала транзакций (глава 11) и системы ввода/вывода (см. ожидания `PAGEIOLATCH`).

Дополнительная информация приведена в главах 3 и 11.

Об авторе

Дмитрий Короткевич — Microsoft Data Platform MVP и Microsoft Certified Master (MCM) с многолетним опытом работы в сфере IT. Он имел дело с Microsoft SQL Server в качестве разработчика приложений и баз данных, администратора и архитектора баз данных. Дмитрий специализируется на проектировании, разработке и наладке сложных систем OLTP, которые круглосуточно обрабатывают тысячи транзакций в секунду. Сейчас он возглавляет группу по обслуживанию баз данных проекта Chewy.com, дает консультации по SQL Server и обучает работе с ним клиентов по всему миру.

Дмитрий регулярно выступает на различных мероприятиях по SQL Server. Он ведет блог на aboutsqlserver.com, иногда пишет в Twitter @aboutsqlserver, и с ним можно связаться по адресу dk@aboutsqlserver.com.

Иллюстрация на обложке

На обложке изображен кроличий сыч (Athene cunicularia). Название рода Athene — это имя древнегреческой богини Афины, одним из традиционных атрибутов которой является сова. Кроличий сыч невелик, но его ноги длиннее, чем у большинства сов. Эти птицы встречаются в Северной и Южной Америке и живут в норах на открытых площадках, а не на деревьях, как большинство сов.

Существует 16 подвидов кроличьего сыча. Внешне они немного различаются, однако у большинства из них коричневые головы и крылья с белыми пятнами, характерные белые брови и белый «нагрудник». Грудь и брюшко совы тоже белые с различными коричневыми узорами в зависимости от подвида. Молодые особи выглядят примерно так же, но без белых пятен. Самцы и самки похожи внешне, но самцы светлее, потому что они проводят больше времени на открытом воздухе в течение дня и оперение выгорает на солнце.

Кроличьи сычи часто гнездятся в норах, выкопанных сусликами. Если найти нору не удается, но почва достаточно мягкая, сыч выкапывает нору сам. Эти сычи обычно создают пары на всю жизнь. Самка откладывает от 4 до 12 яиц в кладке и высиживает их от 3 до 4 недель. После того как птенцы вылупятся, их кормят оба родителя. Сычи питаются в основном мелкими грызунами и крупными насекомыми. В отличие от других сов, они также едят фрукты и семена.

Кроличий сыч считается вымирающим видом в Канаде и находится под угрозой исчезновения в Мексике и некоторых штатах США. Однако он широко распространен в открытых районах Южной Америки, где сохранность этого вида не вызывает опасений. Многие животные, изображенные на обложках изданий O'Reilly, находятся под угрозой исчезновения, и все они важны для мира.

Иллюстрацию на обложке выполнила Карен Монтгомери на основе старинной ксилографии из «Жизни животных» Брема.

Дмитрий Короткевич

SQL Server. Наладка и оптимизация для профессионалов

Перевел с английского Д. Павлов

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Е. Строганова*
Научный редактор	*Р. Чебыкин*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Т. Никифорова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 05.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 28.03.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Тираж 700. Заказ 0000.