**Версионность в Yukon**

Иван Бодягин

**Введение**

Очередную версию СУБД Microsoft SQL Server, являющейся одним из лидеров рынка, ждали довольно долго и, похоже, не зря. В этом продукте только список нововведений будет достаточно увесистым, а уж полное описание новых возможностей потянет на целую книгу. На данный момент доступна лишь альфа-версия продукта, а релиз ожидается примерно через год, но, тем не менее, уже по этой версии можно понять, что нас ожидает в будущем. В данной статье описывается только одно, но достаточно серьезное нововведение, а именно поддержка версионности. Эту функциональность попытались встроить в классический блокирующий сервер (далее – блокировочник), и очень интересно посмотреть, что же из этого получилось.

**Общие принципы**

Параллельное выполнение транзакций способно привести базу в несогласованное состояние даже в тех случаях, когда каждая транзакция, выполняющаяся отдельно от других, производит полностью корректные изменения. Поэтому очередность операций различных транзакций должна тем или иным образом регулироваться.

Во всех предыдущих версиях Microsoft SQL Server механизм подобной регулировки был основан на блокировках. Однако в новой версии (кодовое название Yukon) будет введена поддержка другого механизма, основанного на контроле версий (multiversioning). В дальнейшем два этих подхода я буду называть версионным и блокировочным соответственно.

Версионность сама по себе, не является новым словом в способах обеспечения корректности параллельной обработки транзакций, теоретические работы были еще в начале восьмидесятых, да и появление первых коммерческих реализаций относится примерно к тому же времени. Однако до последнего момента разработчики Microsoft SQL Server последовательно совершенствовали блокировочный механизм, получив в итоге одну из самых удачных реализаций классического блокировочника в индустрии, а вот теперь, похоже, дошли руки и до версионности. Привлекательность данного механизма заключается в том, что читающие запросы никак не мешают пишущим, и наоборот. Но ничто не дается даром. Впрочем, обо всем по порядку.

От механизма параллельной обработки транзакций формальная теория требует одного: чтобы конечный эффект от параллельного выполнения был таким, как будто бы транзакции выполнялись последовательно, при этом порядок их следования фактически не важен. При соблюдении этого условия, если все транзакции сами по себе корректны, то и их параллельное выполнение целостность базы никоим образом не нарушит. Данное условие носит название «критерия упорядоченности».

Способов соблюдения вышеупомянутого критерия существует достаточно много. Помимо уже упоминавшихся механизмов, основанных на блокировках и хранении версий, существует еще чуть ли не десяток других. Более того, существует и формальное доказательство того, что этому критерию можно следовать, используя и комбинированные подходы, применяя разные механизмы для разных типов запросов. Опять-таки известны и весьма успешные практические реализации подобных гибридов.

**Уровни изоляции**

Критерий упорядоченности всем хорош, кроме одного – строгое следование ему слишком дорого обходится с точки зрения производительности. Но, поскольку данный критерий является достаточным, но не необходимым условием корректности параллельной обработки транзакций, то в зависимости от характера транзакций можно повысить степень их вмешательства в работу друг друга без печальных последствий. Чтобы хоть как-то формализовать эти вмешательства, были введены так называемые «уровни изоляции» (Isolation Level).

Если по-простому, то «уровень изоляции» - это степень параллелизма транзакций. В стандарте ANSI SQL уровни изоляции введены посредством феноменов – нежелательных побочных эффектов от излишнего параллелизма, таким образом, что каждый более высокий (более строгий) уровень изоляции устраняет очередной феномен, а также не допускает проявления феноменов, уже устраненных более низким (менее строгим) уровнем.

В стандарте описаны четыре уровня изоляции:

read uncommitted – чтение незафиксированных («грязных») данных. Это самый низкий уровень изоляции. Он лишь гарантирует, что не произойдет феномена «грязной записи» (Dirty Write). Но при этом уровне изоляции вполне возможен феномен «грязное чтение» (Dirty Read). Допустим, первая транзакция записала какие-то данные в X и не зафиксировалась, то есть данные в X не зафиксированы. Если другая транзакция эти данные прочитает, а потом первая будет отменена по какой-то причине, то получится, что вторая транзакция прочитала данные, которые никогда не существовали.

read committed - чтение зафиксированных данных. Гарантируется, что происходит чтение только зафиксированных данных. То есть такого безобразия, как в предыдущем примере, не произойдет. Но зато, если одной транзакции потребуется два раза прочитать некие данные, и между двумя чтениями вклинится другая транзакция, которая успеет эти данные поменять или вставить новые, и зафиксироваться, то два чтения одних и тех же данных в одной транзакции будут отличаться.

repeatable read. Проблему двух последовательных чтений одних и тех же данных этот уровень изоляции решает. Однако если между двумя чтениями в базу добавились новые данные, удовлетворяющие тому же критерию, по которому производилось первое чтение, то эти новые данные во второй раз так же прочитаются. Данный феномен носит название «Фантомное чтение» (Phantom read).

serializability. Самый высокий уровень изоляции. Он не опирается ни на какие феномены, а напрямую формулируется из критерия упорядоченности. То есть при данном уровне изоляции никакие феномены не возможны по определению, так как феномен – это нарушение последовательного выполнения транзакций, что в данном случае невозможно.

Однако за подобную классификацию стандарт подвергался неоднократной и, в общем-то, справедливой критике. Дело в том, что в данную градацию идеально вписываются только чистые блокировочники, но если применяется немного другой способ обеспечения параллелизма, то его уже достаточно проблематично свести к этим четырем уровням, да и не всегда нужно. Другие механизмы обеспечения параллелизма могут допускать другие феномены, быть чуть строже или чуть слабее. По большому счету, неизменным остается лишь одно требование – гарантия упорядоченности, все остальное – серые зоны с нечеткими границами, которые очень сильно зависят от деталей конкретной реализации.

Разберем сначала вкратце классический блокировочник, а затем рассмотрим, что принесла версионность.

**Блокировочный механизм**

Принцип действия, в общем-то, ясен из названия - в основе лежит протокол двухфазной блокировки. Перед чтением или изменением объект (запись) блокируется. То есть другим транзакциям запрещается изменять или даже читать этот объект до тех пор, пока первая транзакция не закончит с ним работать.

Уровень изоляции read committed обеспечивается за счет того, что читающие запросы в транзакциях не удерживают своих блокировок до конца транзакции, а снимают их сразу же, после прочтения. Таким образом, если read committed-транзакция дважды прочитает один и тот же объект, то его значение может отличаться, так как ничто не помешает другой транзакции изменить его в промежутке между двумя чтениями.

При уровне изоляции repeatable read читающие запросы удерживают свои блокировки до конца транзакции, но они блокируют множество реальных записей, существующих на начало транзакции, а не записей, отвечающих условию выборки, которые могут появиться во время жизни транзакции. Например, если выбрать все записи, где x=2, то на всех этих записях будет удерживаться блокировка и поменять их будет нельзя. Но ничто не помешает добавить в другой транзакции еще несколько записей с x=2, и вторая выборка записей с этим же условием в первой транзакции вернет, в том числе, и эти добавленные записи.

Уровень изоляции serializable обеспечивается наложением так называемых предикатных блокировок. Это означает, что блокировка накладывается не только на объект, но и на условие. Если брать предыдущий пример, то мы не сможем добавить запись с x=2 в другой транзакции, если первая сделала выборку по этому условию, так как само условие x=2 оказалось заблокированным. Таким образом, даже повторное чтение любых данных в первой транзакции всегда будет возвращать один и тот же результат.

Более подробно о блокировках можно прочитать в статье «Механизм блокировок Microsoft SQL Server 2000» в третьем номере RSDN Magazine за 2003 год. Сейчас больший интерес представляет версионный механизм.

**Версионный механизм**

Принцип действия версионности основан на том, что транзакция, изменяя данные, порождает новую копию (версию) данных, с которой и работает. Другим транзакциям эта версия не видна, до тех пор, пока первая не зафиксируется. При этом даже после фиксации первой транзакции, устаревшая версия какое-то время сохраняется для корректной работы транзакций, стартовавших до завершения работы первой, но еще не успевших зафиксироваться.

Для читающих запросов все работает очень красиво и эффективно. Они просто получают согласованный срез данных на момент начала транзакции или запроса, но для пишущих запросов и транзакций все не так просто. Если две транзакции решат изменить один и тот же объект, то возникнет конфликт версий. Побеждает та транзакция, которая успела первой, а опоздавшую, как правило, приходится откатывать. С точки зрения производительности откат довольно-таки дорогая операция, к тому же приходится в обязательном порядке предусматривать обработку подобного конфликта. Если в чистом блокировочнике откат транзакции явное следствие ошибки, то в версионнике откат может произойти во вполне невинной ситуации.

Что касается уровней изоляции в версионной модели, то они могут трансформироваться примерно в следующее.

Read uncommitted. В чистом версионнике обычно не реализован, так как «грязные» данные незафиксированных транзакций другим транзакциям не видны, да и не зачем.

Read committed. Этот уровень изоляции несколько строже, чем в модели, основанной на блокировании. В блокировочнике, если читающая транзакция с уровнем изоляции read committed наткнется на уже измененный, но еще не зафиксированный объект, то она будет ожидать его изменения и прочитает уже измененные данные. В версионнике же читающий запрос, как правило, берет версию данных на момент начала запроса, поэтому выборка будет согласованной. При этом не важно, меняются ли данные в настоящий момент какой-либо посторонней транзакцией или нет. Однако повторный запрос тех же данных, если их успели поменять между двумя запросами, вернет уже измененные значения. Поэтому здесь можно наблюдать тот же самый эффект неповторяемого чтения, что и в блокировочном read committed. Изменения же данных при этом уровне изоляции в версионнике, в плане согласованности, мало чем отличаются от аналогичных действий блокировочника, так как изменяться должна все равно последняя зафиксированная версия данных, а не устаревшая копия. Однако небольшие отличия, о которых будет рассказано позднее, все же есть.

Repeatable read. Как правило, версионники этот уровень изоляции не поддерживают, поскольку достаточно легко реализуется более строгий уровень изоляции, который помимо неповторяемого чтения устраняет и появление фантомов.

Snapshot. Этого уровня изоляции нет в классификации ANSI, он присутствует только в версионниках. Суть его в том, что при этом уровне изоляции обеспечивается согласованный срез данных на момент начала транзакции, никакие изменения, произошедшие после старта транзакции в ней не видны. Иными словами, делается то же самое, что и в Read committed, но не для одного запроса, а для транзакции в целом. Таким образом, исключаются и неповторимые чтения, и фантомные чтения. С записью данных при этом уровне изоляции все несколько сложнее, так как если транзакция при изменениях обнаруживает, что необходимые ей данные уже поменял кто-то другой, то ее приходится откатывать.

Serializable. Хотя предыдущий уровень изоляции в версионнике устраняет практически все возможные феномены, но, тем не менее, вероятность неупорядоченности по-прежнему остается, поэтому необходимость в данном уровне изоляции сохраняется. В классическом версионнике упорядоченность достигается за счет комбинации snapshot-уровня изоляции и фиктивного изменения некоторых записей, дабы их не поменяли другие транзакции. В гибридных системах, как правило, можно обойтись и меньшей кровью.

**Реализация в Yukon**

В Yukon версионность не является состоянием сервера в целом, она может быть включена для каждой базы в отдельности, причем по умолчанию версионность включена только для служебных БД master и msdb, и тестовой AdventureWorks.

Версионность включается с помощью нехитрой команды:

|  |
| --- |
| ALTER DATABASE database\_name SET ALLOW\_SNAPSHOT\_ISOLATION ON |

После ее выполнения сервер не сразу переключает базу в версионный режим, а переводит механизм поддержки версионности (snapshot isolation framework) в состояние PENDING\_ON, поскольку в этот момент в базе могут быть активные транзакции. После завершения всех активных транзакций над базой производятся все необходимые изменения, механизм версионности для нее переводится в состояние ON, и появляется возможность выполнять версионные запросы. Обратное действие осуществляется также в два этапа, сначала БД переводится в состояние PENDING\_OFF, а потом уже отключается механизм поддержки версионности.

Проверить состояние всех баз на сервере можно с помощью вот такого запроса:

|  |
| --- |
| SELECT name, snapshot\_isolation\_framework FROM sys.databases |

Здесь «snapshot\_isolation\_framework» может принимать следующие значения: 0 – OFF,1 – ON, 2 – PENDING\_OFF, 3 – PENDING\_ON.

Механизм поддержки версионности отвечает за сохранение копий данных, выборку нужных версий при версионных запросах, автоматическое определение конфликтов при изменениях и выполняет остальную рутинную работу по поддержке версионности. Он является не отдельной службой или модулем, а относительно простым набором алгоритмов, которые включаются в работу при переводе базы в соответствующее состояние.

Физически механизм версионности в Yukon отслеживает изменение данных отдельно для каждой записи. Для этого к записи добавляется кусочек длиной в 14 байт, в котором хранится идентификатор транзакции, изменившей эту запись, и ссылка на предыдущую зафиксированную версию. Таким образом, все версии одной записи образуют связный список, и серверу в случае необходимости не составляет никакого труда спуститься по цепочке ссылок к нужным данным.

|  |
| --- |
| ПРИМЕЧАНИЕ  В том случае, если на странице данных не хватает места под дополнительные 14 байт на каждую запись, то во время перевода базы в «версионное» состояние данные будут автоматически перераспределены по страницам. |

Все версии записи собираются в специальном хранилище (version store heaps), которое расположено в tempdb. При любом изменении записи предыдущая версия копируется в это хранилище.

Благодаря особенностям tempdb и отсутствию журналирования для хранилища версий при обслуживании и чтении копий данных, нагрузка на операции ввода/вывода обещает быть минимальной.

Те устаревшие копии данных, которые уже точно не будут нужны ни одной транзакции, удаляются из tempdb автоматически. При этом отслеживается старейшая заинтересованная транзакция. Поэтому исключается вероятность того, что сервер в какой-то момент не найдет нужную версию. Удаление производит специальный механизм, который учитывает текущую нагрузку, через количество активных версионных транзакций, и процент свободного места в tempdb.

Для оценки предполагаемого размера хранилища версий предлагается следующая формула: Size (KB) = частота обновления версий (Version generation rate) Kb/sec. \* время выполнения самой длинной транзакции (the longest transaction time) sec.

**Как это работает**

Здесь опять-таки проще всего разобрать по очереди все уровни изоляции предлагаемые Yukon снизу вверх.

Read uncommitted

В чистом версионнике, как уже говорилось, read uncommitted обычно не предусмотрен, да и не нужен. В Yukon при запросах к БД с включенной поддержкой версионности такой фокус тоже провернуть не получится. Даже при явном указании соответствующих подсказок оптимизатору в запросе, возвращается предыдущая зафиксированная версия, а не грязные данные транзакции, заблокировавшей запись от изменений.

Read committed

С read committed все несколько сложнее. После включения поддержки версионности все запросы на чтение при этом уровне изоляции автоматически начинают работать как версионные. То есть если транзакция при чтении натыкается на заблокированную запись, то она читает предыдущую версию этой записи из tempdb.

|  |
| --- |
| ПРИМЕЧАНИЕ  На самом деле в alpha-версии Yukon для поддержки версионности при read committed необходимо включить специальный трейс-флаг (3970). Но Microsoft торжественно клянется, что в финальном продукте все будет происходить автоматически. |

Можно провести простенький эксперимент. Пусть есть небольшая табличка tst в БД с поддержкой версионности, например, AdventureWork, созданная с помощью вот такого скрипта:

|  |
| --- |
| CREATE TABLE tst(x int, y int)  GO  INSERT INTO tst(x, y) VALUES(1, 5)  INSERT INTO tst(x, y) VALUES(2, 4)  INSERT INTO tst(x, y) VALUES(3, 3)  INSERT INTO tst(x, y) VALUES(4, 2)  INSERT INTO tst(x, y) VALUES(5, 1) |

Сначала откроем новое подключение, откроем read committed-транзакцию и сделаем выборку, транзакцию при этом закрывать не будем.

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL READ COMMITTED  BEGIN TRAN  SELECT \* FROM tst WHERE x = 3 |

Получим то, что и ожидалось: x = 3, y = 3.

Теперь в другой транзакции попытаемся обновить эту запись, также не фиксируя транзакцию.

|  |
| --- |
| BEGIN TRAN  UPDATE tst SET y = -1 WHERE x = 3 |

Если после этого взглянуть на блокировки, наложенные на табличку tst, то, как и при использовании предыдущих версий SQL Server, можно заметить эксклюзивную блокировку на запись и две блокировки намерения выше по иерархии, на страницу и таблицу.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тип | Описание | Объект | Режим | Статус | spid |
| TAB |  | 1963154039 | IX | GRANT | 52 |
| RID | 1:1357:2 | 72057594057326592 | X | GRANT | 52 |
| PAG | 1:1357 | 72057594057326592 | IX | GRANT | 52 |

Таблица 1

То есть картина совпадает с той, которую можно видеть при использовании предыдущей версии SQL Server или БД без поддержки версионности.

Однако дальше начинаются отличия. Если сейчас вернуться к первой транзакции и попытаться опять выполнить тот же самый запрос, то он совершенно спокойно отработает.

|  |
| --- |
| SELECT \* FROM tst WHERE x = 3 |

И результат будет точно таким же: x=3, y=3. Если попробовать сделать то же самое на БД без включенной поддержки версионности, то второй запрос из первой транзакции не выполнится. Он будет ожидать фиксации или отката второй транзакции. То есть он попросту не сможет прочитать нужную запись, поскольку она заблокирована. Но в данном случае блокировка нисколько не мешает прочитать версию данных, существовавшую на момент начала выборки.

Более того, в силу особенностей работы с неиндексированными таблицами (а для тестовой таблицы индексов не создавалось), в базе без поддержки версионности второй запрос в первой транзакции не смог бы выбрать не только заблокированную запись, но и любую другую. Из-за блокировки ему все равно бы пришлось ждать завершения работы первой транзакции. Говоря проще, у блокировочника, в случае отсутствия индексов, блокировка одной записи превращается фактически в блокировку всей таблицы.

Если теперь зафиксировать изменения тестовой таблицы, произведенные второй транзакцией:

|  |
| --- |
| COMMIT TRAN |

и сделать опять выборку тех же данных в транзакции номер один:

|  |
| --- |
| SELECT \* FROM tst WHERE x = 3 |

то мы уже получим результат x = 3, y = (-1), феномен неповторимого чтения (non-repeatable read) в действии. Нам удалось дважды обратиться к одним и тем же данным из одной транзакции и получить различные результаты.

Итак, для читающих запросов все работает просто здорово, перейдем теперь к пишущим запросам все при том же уровне изоляции.

Допустим, версионник пытается обновить несколько записей из read committed-транзакции и натыкается на то, что кто-то в данный момент из другой транзакции обновил, но еще не зафиксировал одну из нужных записей. Просто подождать, пока вражеская транзакция зафиксируется, и затем поменять запись, нельзя, так как к этому времени запись может быть изменена таким образом, что перестанет удовлетворять критериям запроса. Чтобы этого не произошло, необходимо, как минимум, перечитать эту запись заново - чтобы изменять актуальную версию, а не устаревшую.

Таким образом возникает некоторый парадокс – при одном и том же уровне изоляции читающие запросы получаются согласованнее, чем пишущие. Формально все в порядке – требования уровня изоляции не нарушены. Чтобы избежать такого парадокса, в некоторых коммерческих реализациях в таких случаях делается откат запроса, а затем запрос выполняется заново, чтобы обеспечить обновление на согласованном срезе данных.

Разработчики Yukon не стали добавлять таких сложностей, и все изменения делаются по старинке, точно так же, как их делает обычный блокировочник. Вплоть до побочного эффекта, связанного с блокированием всей таблицы по причине отсутствия индекса.

Если, например, в одном подключении выполнить часть транзакции, изменив в тестовой табличке одну запись, но не фиксировать транзакцию, удерживая тем самым блокировку…

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL READ COMMITTED  BEGIN TRAN  UPDATE tst SET y=3 WHERE x=3 |

А в другой транзакции попытаться изменить другую запись…

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL READ COMMITTED  BEGIN TRAN  UPDATE tst SET y=-1 WHERE x=4  COMMIT |

То вторая транзакция не завершится, а подвиснет в задумчивости. Если же в этот момент взглянуть на блокировки, наложенные на таблицу tst:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тип | Описание | Объект | Режим | Статус | spid |
| PAG | 1:1357 | 72057594057326592 | IU | GRANT | 54 |
| PAG | 1:1357 | 72057594057326592 | IX | GRANT | 53 |
| RID | 1:1357:2 | 72057594057326592 | X | GRANT | 53 |
| RID | 1:1357:2 | 72057594057326592 | U | WAIT | 54 |
| TAB |  | 1963154039 | IX | GRANT | 54 |
| TAB |  | 1963154039 | IX | GRANT | 53 |

Таблица 2

то можно наблюдать картину, характерную для самого обычного блокировочника, хотя версионности никто не отменял, в чем можно убедиться, выполнив в еще одной параллельной read committed-транзакции читающий запрос:

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL READ COMMITTED  BEGIN TRAN  SELECT \* FROM tst  COMMIT |

Этот запрос совершенно спокойно выполнится, никого не потревожив, и вернет при этом состояние таблицы на момент, предшествующий изменениям, так как ни одна из изменяющих таблицу транзакций на время выполнения этого запроса еще не зафиксирована. Все дело в том, что обычно версионник выполняет изменения данных примерно по такому сценарию:

Делается версионное сканирование таблицы, и выясняется, какие записи необходимо изменить.

Предпринимаются попытки изменить отобранные записи.

Если запись изменилась с момента версионного скана, то проверяется, не перестала ли она удовлетворять критерию отбора, и если не перестала, то запись меняется, если перестала, то пропускается. (Некоторые реализации применяют здесь более хитрые алгоритмы, но этот вопрос выходит за рамки данной статьи).

Если запись меняется в настоящий момент, то сервер дожидается конца изменений и опять-таки проверяет соответствие записи условиям выборки.

То есть ожидание , если и происходит, то только из-за того, что в процессе обновления встретилась запись, которая меняется в данный момент.

Чистый же блокировочник работает немного по другому сценарию. Сканирование данных ему не имеет смысла делать, так как все запросы на чтение у него блокирующие. Поэтому он просто перебирает все записи в таблице по очереди (напомню, речь идет о таблице без индексов), проверяя их на соответствие условию выборки, и накладывая при этом блокировку обновления (update lock). Такая блокировка совместима с блокировками чтения, но несовместима сама с собой и с монопольными блокировками. Таким образом, читающим запросам подобный перебор не мешает, но другие блокировки обновления и монопольные будут помехой этому запросу. Следовательно, если в момент перебора в таблице монопольно заблокирована хотя бы одна запись (что и имеет место в данном примере, так как запись была изменена, но транзакция еще не зафиксирована), то рано или поздно изменяющий запрос до нее доберется и зависнет на блокировке, ожидая фиксации «вражеской» транзакции.

Несмотря на возможность версионных запросов, Yukon все равно при записи данных поступает как блокировочник, что и приводит к вышеописанному эффекту.

Repeatable read

Уровень изоляции repeatable read в базе с включенной поддержкой версионности работает точно так же, как и на базе без оной. Совершенно спокойно накладываются и удерживаются должное время все положенные по статусу разделяемые (share) блокировки. Да в общем-то, вряд ли тут вообще что-то могло измениться. Но появилась одна полезная возможность: Если запрос выполняется по базе с включенной поддержкой версионности, то при указании оптимизатору хинта READCOMMITTED в читающем запросе, выборка будет версионной. Возможность действительно довольно полезная - в связи с некоторыми особенностями уровня изоляции snapshot.

Snapshot

Уровень изоляции snapshot, является чисто версионным, в отличие от предыдущего, чисто блокировочного, и вообще совершенно новым для SQL Server.

Читающие запросы при этом уровне изоляции выполняются так, как и положено им выполняться в честном версионнике при этом уровне изоляции. Если вернуться все к той же тестовой табличке и в одном из подключений начать транзакцию, в которой изменить какую-нибудь запись, но саму транзакцию не фиксировать…

|  |
| --- |
| BEGIN TRAN  UPDATE tst SET y = 2 WHERE x = 4 |

А в другом подключении начать snapshot транзакцию с читающим запросом к той же табличке…

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL SNAPSHOT  BEGIN TRAN  SELECT \* FROM tst |

То snapshot-транзакция, как, впрочем, и версионный read committed, совершенно спокойно отработает, вернув предыдущее значение измененной записи. Однако если сейчас зафиксировать первую, изменяющую транзакцию…

|  |
| --- |
| COMMIT TRAN |

А затем повторить ту же самую выборку из snapshot транзакции…

|  |
| --- |
| SELECT \* FROM tst |

То эта выборка вернет все еще старые значения записей, существовавшие до фиксации первой транзакции. То есть здесь чтение полностью воспроизводимо, в отличие от read committed. Более того, этот уровень изоляции не допускает появления фантомов, в отличие от блокировочного repeatable read. Например, если выполнить третью транзакцию, в которой в таблицу добавляются записи…

|  |
| --- |
| BEGIN TRAN  INSERT INTO tst (x, y) VALUES (6, 0)  COMMIT TRAN |

То очередная выборка всех записей таблицы tst из snapshot-транзакции вернет все те же записи, что и в первый раз, просто потому, что на момент первого запроса нового значения еще не было. В общем, с таблицей можно делать все что угодно, все выборки из snapshot-транзакции будут одними и теми же.

Таким образом, чтение при уровне изоляции snapshot в Yukon практически ничем не отличается от обычного версионного. Оно работает по тем же принципам, и «внешний» эффект точно такой же. Транзакция получает согласованный срез данных, начиная с первого обращения к данным, и все последующие изменения ее не касаются.

Все это очень хорошо работает при читающих запросах, однако при записи могут возникать конфликты. Если при выполнении обновления snapshot-транзакция доберется до записи, заблокированной другой транзакцией, то, возникнет конфликт версий. Если блокирующая транзакция успешно фиксируется, в чистом версионнике snapshot-транзакция откатывается, поскольку если она изменит данные более «молодой» транзакции и продолжит работу, вполне возможен феномен утерянного обновления. Сместить эти транзакции друг относительно друга во времени и считать snapshot-транзакцию более «молодой» тоже не получится, так как блокирующая транзакция могла добавить записи, удовлетворяющие условию выборки snapshot-транзакции, а значит, все snapshot-запросы должны были эти записи увидеть. То есть snapshot-транзакция все равно должна выполниться заново, с более поздней временной меткой, чтобы увидеть все изменения, внесенные блокирующей транзакцией.

И в данном случае Yukon мало чем отличается от версионника. Если при уровне изоляции read committed в случае изменения он может себе позволить вести себя как блокировочник, то при уровне изоляции snapshot такой фокус не пройдет. Как минимум при этом snapshot скатится все до того же read committed. Дело в том, что блокировочник уровни изоляции выше read committed обеспечивает удержанием коллективных (share) блокировок при запросах на чтение до конца транзакции. Версионник же подобных блокировок для обеспечения уровня изоляции snapshot не использует, у него принцип совсем другой. А поскольку к моменту конфликта snapshot-транзакция уже могла выполнить несколько версионных чтений, то поступать как блокировочник уже поздно, поэтому Yukon, так же как и версионник, в случае конфликта откатывает snapshot-транзакцию.

Если продолжить издевательства над таблицей tst, и изменить в ней какую-нибудь запись, не фиксируя транзакцию:

|  |
| --- |
| BEGIN TRAN  UPDATE tst SET y = -1 WHERE x = 3 |

а потом попробовать изменить ту же запись из snapshot-транзакции:

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL SNAPSHOT  BEGIN TRAN  UPDATE tst SET y = 3 WHERE x = 3  COMMIT TRAN |

То snapshot-транзакция повиснет в задумчивости. При взгляде на блокировки, наложенные на таблицу tst, можно наблюдать картину, характерную для обычного блокировочика.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тип | Описание | Объект | Режим | Статус | spid |
| TAB |  | 1963154039 | IX | GRANT | 51 |
| RID | 1:1357:2 | 72057594057326592 | U | WAIT | 51 |
| PAG | 1:1357 | 72057594057326592 | IU | GRANT | 51 |
| TAB |  | 1963154039 | IX | GRANT | 52 |
| RID | 1:1357:2 | 72057594057326592 | X | GRANT | 52 |
| PAG | 1:1357 | 72057594057326592 | IX | GRANT | 52 |

Таблица 3

Snapshot-транзакция (spid 52) ожидает на блокировке (U – WAIT), пока освободится нужная запись (RID 1:1357:2), заблокированная другой транзакцией (spid 51) монопольно (X - GRANT).

Если сейчас вернуться в первое окно и откатить блокирующую транзакцию, то snapshot совершенно спокойно выполнит свое обновление и зафиксируется. Однако если блокирующую транзакцию зафиксировать, то, в отличие от блокировочного поведения, snapshot-транзакция будет отменена, и клиентское приложение получит сообщение об ошибке:

|  |
| --- |
| .Net SqlClient Data Provider: Msg 3960, Level 16, State 1, Line 1  Cannot use snapshot isolation to access table 'tst' in database 'AdventureWorks'.  Snapshot transaction aborted due to update conflict. Retry transaction. |

Более того, для отката snapshot-транзакции ей даже не нужно ожидать снятия блокировки. Чтобы избежать несогласованного изменения, необходимо производить откат даже в том случае, если после старта snapshot-транзакции одна из записей, необходимых для пишущего запроса, была изменена другой транзакцией, успевшей зафиксироваться.

Если в одном из подключений начать snapshot-транзакцию, сделав простую выборку:

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL SNAPSHOT  BEGIN TRAN  SELECT \* FROM tst |

Затем в другом подключении изменить какую-нибудь запись:

|  |
| --- |
| BEGIN TRAN  UPDATE tst SET y=3 WHERE x=3  COMMIT TRAN |

А потом попытаться изменить эту же запись из snapshot-транзакции:

|  |
| --- |
| UPDATE tst SET y=3 WHERE x=3  COMMIT TRAN |

То snapshot транзакция успехом не увенчается, а будет отменена по причине конфликта версий.

Дело в том, что если существует транзакция, зафиксированная после начала работы snapshot-транзакции, и изменения этой транзакции попадают в условия выборки, сделанной snapshot-транзакцией, то вполне могут существовать записи, которые были изменены этой транзакцией после того, как snapshot-запрос их просмотрел, таким образом, что они стали удовлетворять условию snapshot-выборки. То есть получается, что snapshot-транзакция может увидеть только часть изменений, сделанных другой транзакцией, а этого допустить нельзя.

В данном случае поведение Yukon очень похоже на версионное. Но на самом деле он все равно ведет себя немного «по-блокировочному». Проявляется это, например, все в той же ситуации с отсутствием индексов.

Если еще раз выполнить блокирующую транзакцию, не фиксируя ее:

|  |
| --- |
| BEGIN TRAN  UPDATE tst SET y=3 WHERE x=3 |

А потом попробовать изменить другую запись из snapshot транзакции:

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL SNAPSHOT  BEGIN TRAN  UPDATE tst SET y=-1 WHERE x=4  COMMIT TRAN |

То snapshot транзакция все равно окажется заблокированной.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тип | Описание | Объект | Режим | Статус | spid |
| RID | 1:1357:2 | 72057594057326592 | U | WAIT | 51 |

Таблица 4

И, что характерно, блокировка произойдет на той же самой записи. Иными словами, в отсутствии индексов Yukon осуществляет поиск по таблице как самый обычный блокировочник, даже при уровне изоляции snapshot. Версионник в данной ситуации сначала сканирует всю таблицу обычным версионным запросом, выясняя нужные записи, а потом обращается с обновлением только к тем, которые подходят под условие выборки, поэтому подобного там не происходит. Yukon же сканирует таблицу, накладывая блокировку обновления на каждую запись, и снимая ее, если запись не удовлетворяет условию выборки, или конвертируя в эксклюзивную, если удовлетворяет. Таким образом, даже если заблокированная запись не удовлетворяет условию выборки обновляющего оператора snapshot-транзакции, в случае отсутствия индексов транзакция все равно застрянет на блокировке.

Результат подобного ожидания snapshot-транзакции вполне закономерен. Если блокирующую транзакцию зафиксировать, то snapshot-транзакция будет отменена, несмотря на то, что формально измененная запись ни до, ни после запроса под условия выборки не попадала. Оборачивается это тем, что если в момент изменения записей в таблице без индексов из snapshot-транзакции будет изменена любая запись, то snapshot-транзакция не выполнится.

Судя по всему, сделать согласованное неблокирующее сканирование при обновлении записей – задача нетривиальная. Поэтому все сделано единообразно, «по-блокировочному». С одной стороны, подобное блокировочное сканирование обещает довольно мрачные перспективы попыткам записи из snapshot-транзакций в неиндексированные таблицы. Но, с другой стороны, это предотвращает некоторые экзотические случаи фантомов, которые пропускает классический snapshot.

Serializable

Несмотря на то, что версионный snapshot помогает избавиться от большинства побочных эффектов, связанных с параллельным выполнением транзакций, вероятность появления некоторых феноменов по-прежнему остается. Чтобы избавиться от подобных эффектов, при работе с классическим версионником и некоторыми гибридами приходится предпринимать дополнительные меры в виде фиктивных изменений или специальных блокировок некоторых записей.

Но Yukon, как и предыдущие версии SQL Server от Microsoft, поддерживает уровень изоляции serializable, который по определению не допускает никаких феноменов. По механизму этот уровень изоляции является чисто блокировочным и никакие версионные запросы, даже на чтение, здесь не поддерживаются, если конечно, не давать специальных указаний оптимизатору.

Заключение

Новая функциональность, безусловно, окажется очень полезной. Самый заметный эффект – это отсутствие блокировок между читающими и пишущими запросами. То есть читающие запросы не блокируют пишущие, и наоборот. Собственно, вся версионность ради этого и затевалась.

У чистых блокировочников, каковым до недавнего времени являлся и Microsoft SQL Server, при наступлении того печального момента, когда читающие запросы начинают довольно сильно конфликтовать с пишущими, используется стандартный архитектурный прием. Механизм работы с данными делится на две составляющие, OLAP и OLTP. OLTP реализует работу пишущих транзакций на относительно небольшом объеме актуальных данных, а OLAP содержит основные данные, доступные только для чтения, причем в силу того, что каждый механизм оптимизирован исключительно под свои задачи, подобное решение, как правило, оказывается очень эффективным. При этом совершенно необязательно сразу строить большую систему и покупать дополнительное оборудование. Если задача не требует излишней громоздкости, то начинается все обычно с построения промежуточных агрегирующих таблиц и материализованных представлений, в дальнейшем возможен перенос одной из составляющих в отдельную базу, экземпляр сервера и, наконец, если в том возникнет необходимость, на отдельный сервер или даже группу серверов.

Может показаться, что класс задач, для которых реально нужна версионность, достаточно узок. С одной стороны, использование версионности имеет смысл, если в блокировочнике конфликт читающих и пишущих запросов начинает оказывать заметное влияние на производительность. Но с другой стороны, при дальнейшем увеличении нагрузки, производительность версионного механизма довольно быстро начнет снижаться из-за частых обновлений по причине больших накладных расходов на поддержку версионности, она ведь тоже обходится не даром.

Но тут можно вспомнить, что существуют задачи, где необходимо выполнять большое количество малопрогнозируемых запросов (ad-hoc queries). В этом случае конфликты читающих и пишущих запросов выходят на первое место, поскольку в подобной ситуации очень высока вероятность возникновения взаимоблокировок.

Также стоит заметить, что, несмотря на всю полезность агрегатных таблиц, они обладают одним большим недостатком, сильно сужающим область их применения. Если в исходной таблице активность запросов можно физически разнести по разным страницам данных, то в агрегатной таблице большое количество изменений создаст совершенно ненужный ажиотаж в очень маленьком объеме.

Немаловажно также и то, что писать приложения для БД с поддержкой версионности попросту удобнее. Отсутствие необходимости отслеживать возможные конфликты читающих и пишущих запросов здорово облегчает жизнь, особенно не слишком опытным разработчикам, и повышает масштабируемость системы (хотя в этом случае выше вероятность появления некоторых неочевидных эффектов).

И наконец, введенная поддержка версионности позволит легче адаптироваться к новому серверу тем, кто привык работать с подобной функциональностью.

Преимущества версионности.

Читающие запросы не блокируют пишущие, и наоборот.

Уменьшатся вероятность возникновения взаимоблокировок.

Есть возможность выполнять большие согласованные чтения, не предусмотренные структурой базы (ad-hoc queries), без риска сильного падения производительности остальных запросов.

Можно выполнять согласованные агрегатные запросы без повышения уровня изоляции.

Во что обходится версионность.

Пишущие транзакции обязаны создавать копию записи в хранилище версий, даже если на данный момент нет читающих запросов, которым нужна эта запись.

Под хранилища версий, которые расположены в tempdb, необходимо дополнительное место. Причем его может потребоваться довольно много (при наличии длинных версионных транзакций и большом количестве изменений).

Длина каждой записи увеличивается на 14 байт.

Читающие версионные запросы выполняются тем дольше, чем старее транзакция, из-за необходимости спускаться по цепочке ссылок к нужной версии данных. Это также порождает дополнительную нагрузку на процессор и память.

Часть пишущих транзакций при snapshot-уровне изоляции может быть отменена из-за конфликта версий при изменении данных.

В целом реализация версионности от Microsoft выглядит довольно удачно. Существует широкий класс задач, для которых выгоднее использовать чисто блокировочный механизм. Поэтому возможность отключить совершенно не нужную в данном случае версионность довольно полезна.

При включенной поддержке версионности становятся доступны практически все прелести версионных СУБД, а поскольку при записи в большинстве случаев серверу выгоднее вести себя как блокировочнику, то Yukon так и поступает, благо опять-таки есть такая возможность. А если из repeatable read- или даже serializable-транзакции понадобиться забрать согласованные данные, никого не блокируя, то можно использовать версионное чтение.

Вообще, с данной точки зрения, очень полезной была бы функциональность позволяющая включать или выключать поддержку версионности отдельно для каждой таблицы. Но здесь есть ряд концептуальных проблем, например, сложно представить, как будет выглядеть объединение в запросе двух таблиц, одна из которых поддерживает версионность, а другая – нет.

Сам механизм обеспечения версионности в Yukon также обещает быть довольно быстрым и надежным. Используемый алгоритм относительно прост и обеспечивает стопроцентное обнаружение конфликтов при версионных изменениях, без холостых срабатываний. Впрочем, о производительности данного механизма можно будет говорить только после проведения относительно адекватных тестов, а еще лучше - после применения в реальном приложении.