**Блокировки в MS SQL Server 2000**

Алексей Ширшов

**Введение**

Обычно блокировки рассматривают совместно с транзакциями. В данной статье упор делается в основном на механизм блокировок, его внутреннее устройство и использование в СУБД MS SQL Server 2000. Предполагается, что читатель хорошо знаком с транзакциями и их свойствами. Давайте вспомним вкратце, какими свойствами должны обладать транзакции в современных СУБД (эти требования носят название ACID – Atomicity, Consistency, Isolation и Durability):

Atomicity (атомарность). Это требование заключается в том, что все данные, с которыми работает транзакция, должны быть либо подтверждены (commit), либо отменены (rollback). Не должно быть ситуации, когда часть изменений подтверждается, а часть – отменяется. Это правило автоматически выполняется для простых данных.

Consistency (согласованность). После выполнения транзакции все данные должны остаться в согласованном состоянии. Другими словами, транзакция либо не изменит данных, и они останутся в прежнем состоянии, либо измененные данные будут удовлетворять ограничениям целостности, правилам (rules) и другим критериям согласованности данных.

Isolation (изолированность). Транзакции должны выполнятся автономно и независимо от других транзакций. При одновременном выполнении множества конкурирующих друг с другом транзакций, любое обновление определенной транзакции будет скрыто от остальных до тех пор, пока эта транзакция не будет зафиксирована. Существуют несколько уровней изолированности (изоляции) транзакций, которые позволяют выбрать наиболее оптимальное решение с точки зрения производительности и целостности данных. Основным методом реализации этих уровней и являются блокировки, о которых пойдет речь в этой статье.

Durability (долговечность или устойчивость). Это требование заключается в том, что после подтверждения изменения данных (фиксации транзакции) система переходит в новое состояние и возврат к старому состоянию не возможен, т.е. не возможен откат в предыдущее состояние. Для вложенных транзакций это требование нарушается.

В данной статье рассматриваются механизмы реализации уровней изолированности транзакции. Стандартом ANSI были определены четыре уровня изоляции транзакций. Первый – это нулевой уровень изоляции, второй – первый уровень и так далее. Эти уровни помогают решать различные проблемы, которые будут рассматриваться подробно далее в процессе написания демонстрационной программы на С++. Определения уровней будут даны в конце раздела.

Итак, чтобы лучше понять проблемы изоляции транзакций, рассмотрим их сначала с точки зрения программирования на С++. Так как наша программа будет оперировать простыми данными (значение типа int), будем считать, что требования атомарности выполняются автоматически. Кроме того, мы не будем налагать каких-либо логических ограничений на значение переменной и не будем использовать вложенных транзакций, так что требования согласованности и устойчивости также будут опущены.

Наша программа содержит всего два класса: CObject и CProxy. Класс CObject - это объект-одиночка (singleton), который содержит переменную value (доступ к этой переменной мы и будем защищать), и некоторый набор служебных функций. Класс CProxy представляет собой посредника для объекта CObject; именно с ним будет работать клиент. Вот первоначальный набросок (в классе CProxy используется нестандартная конструкция \_\_declspec(property), поддерживаемая только компиляторами от Microsoft):

|  |
| --- |
| class CObject;class CProxy{ friend class CObject;public: \_\_declspec(property(get=get\_Value,put=put\_Value)) int value; int get\_Value(int level = -1) const; void put\_Value(int i); void Commit(); void Rollback();private:  int \_level; int \_value; bool fUpd; CProxy(CObject\* par,int level) { fUpd = false; parent = par; \_level = level; } CObject\* parent;};class CObject{ friend class CProxy;public: static CProxy& GetObject(int level = -1); ~CObject() { if (hMutex) CloseHandle(hMutex); }protected: CProxy& BeginTran(int level) { return \*(new CProxy(this,level)); } void RequestExclusive(int level) { }  void RequestShared(int level) { } void RemoveShared(int level) { } void RemoveLocks() { }private: CObject() { value = 0; } int value; static HANDLE hMutex;};\_\_declspec(selectany) HANDLE CObject::hMutex = NULL;CProxy& CObject::GetObject(int level){ HANDLE hLocMutex = CreateMutex(NULL,TRUE,\_T("Guard-Lock-Mutex")); bool flg = GetLastError() == ERROR\_ALREADY\_EXISTS;  if (flg) WaitForSingleObject(hLocMutex,INFINITE); else CObject::hMutex = hLocMutex; static CObject obj; ReleaseMutex(hLocMutex);  if (flg) CloseHandle(hLocMutex); return obj.BeginTran(level);}void CProxy::Commit(){ parent->RemoveLocks(); delete this;}void CProxy::Rollback(){ if (fUpd) parent->value = \_value; parent->RemoveLocks(); delete this;}void CProxy::put\_Value(int i){ parent->RequestExclusive(\_level); if (!fUpd) \_value = parent->value; parent->value = i; fUpd = true;}int CProxy::get\_Value(int level) const{ if (level == -1)  level = \_level; parent->RequestShared(level); int v = parent->value; parent->RemoveShared(level); return v;} |

Клиент никогда не имеет дела непосредственно с экземпляром класса CObject. Экземпляры класса CProxy – представляют копию данных объекта CObject и делегируют запросы на чтение и запись переменной value. Код получился несколько громоздким: к чему такие сложности? Я заранее определил довольно широкий интерфейс, чтобы потом меньше исправлять. :)

Прошу обратить внимание на довольно сложный механизм создания экземпляра CObject в функции GetObject. Обычно в программах используется более простой код, типа:

|  |
| --- |
| CProxy& CObject::GetObject(int level){ static CObject obj; return obj.BeginTran(level);} |

Чем он плох? Дело в том, что если несколько потоков попытаются одновременно вызвать функцию GetObject, конструктор класса CObject может быть вызван более одного раза, так как компилятор (возможно, это его ошибка) не генерирует безопасный код проверки с использованием ассемблерной инструкции cmpxchg. Хотя вероятность возникновения такой ситуации довольно низка, я рекомендую все же не игнорировать ее. Самое простое решение проблемы заключается в использовании недорогого ресурса критической секции, например, так:

|  |
| --- |
| CProxy& CObject::GetObject(int level){ ::EnterCriticalSection(&g\_cs); static CObject obj; ::LeaveCriticalSection(&g\_cs); return obj.BeginTran(level);} |

Однако встает вопрос: где ее инициализировать? Можно в конструкторе глобального объекта, но если у нас будет такой же глобальный клиент, мы не сможем гарантировать, что инициализация критической секции произойдет раньше вызова функции GetObject. Нам нужно что-то, что создается, инициализируется и захватывает ресурс непосредственно в функции GetObject. В качестве этого «чего-то» я выбрал объект исполнительной системы «Мьютекс». Его использование вы и можете наблюдать в первоначальном коде.

Теперь рассмотрим пример с использованием этих классов, который сразу вскрывает первую проблему.

|  |
| --- |
| unsigned \_\_stdcall thread\_proc(void\*){// Начало транзакции CProxy& prx = CObject::GetObject(); prx.value = 20; prx.Commit(); return 0;}int main(int argc, char\* argv[]){// Начало транзакции CProxy& prx = CObject::GetObject(); prx.value = 10;// Начало новой сессии \_beginthreadex(0,0,thread\_proc,0,0,0);// Эмулируем работу// Sleep(1000); printf("%d\n",prx.value); prx.Commit(); return 0;} |

Здесь я в двух параллельных потоках изменяю значение переменной value объекта CObject: в одном – на 10, во втором – на 20. Что выведется на консоль? Определенно сказать нельзя: если раскомментировать строчку Sleep(1000), выведется 20. С закомментированной строчкой выводится 10. Эта проблема носит название «проблема потери последнего изменения» (lost update problem) или проблема «грязной» записи. Она заключается в том, что при одновременном выполнении транзакций, в которых производится изменение данных, невозможно сказать заранее, какое конечное значение примут данные после фиксирования обеих транзакций. В случае «грязной» записи только одна из всех параллельно выполняющихся транзакций будет работать с действительными данными, остальные – нет. Другими словами, хотя данные и будут находиться в согласованном состоянии, логическая их целостность будет нарушена.

Для того чтобы наш объект удовлетворял первому уровню изоляции транзакций, на котором запрещается «загрязнение» данных, перепишем его следующим образом (изменения касаются только класса CObject):

|  |
| --- |
| class CObject{ friend class CProxy;public: enum {READ\_UNCOMMITTED}; static CProxy& GetObject(int level = -1); ~CObject() { DeleteCriticalSection(&exclusive); if (hMutex) CloseHandle(hMutex); }protected: CProxy& BeginTran(int level) { return \*(new CProxy(this,level)); } void RequestExclusive(int level) { if (level >= READ\_UNCOMMITTED) TestExclusive(); }  void RequestShared(int level) { } void RemoveShared(int level) { } void RemoveLocks() { RemoveAllLocks(); }private: CObject() { value = 0; InitializeCriticalSection(&exclusive); } void TestExclusive() { //Проверка на монопольную блокировку EnterCriticalSection(&exclusive); //Вошли больше одного раза if (exclusive.RecursionCount > 1) LeaveCriticalSection(&exclusive); } void RemoveAllLocks() { //Если была установлена монопольная блокировка - снимаем if (exclusive.OwningThread == (HANDLE)GetCurrentThreadId()) LeaveCriticalSection(&exclusive); } int value; CRITICAL\_SECTION exclusive; static HANDLE hMutex;}; |

Добавленный код выделен. Хочется пояснить одну деталь: так как изменять значение переменной value можно несколько раз, а Commit (или Rollback) вызывать только раз, налицо тот факт, что функция EnterCriticalSection вызывается бОльшее количество раз, нежели LeaveCriticalSection. Это ошибка – в соответствии с документацией количество вызовов функций EnterCriticalSection и LeaveCriticalSection должно совпадать. Поэтому после входа в критическую секцию я проверяю поле RecursionCount, которое устанавливает количество повторных входов потока в критическую секцию.

|  |
| --- |
| ПРИМЕЧАНИЕПри работе под ОС семейства Windows 9x это поле не используется и всегда содержит 0, так что приводимый здесь и далее код будет работать только на операционных системах семейства NT. |

Теперь можно определенно сказать, что выведется на консоль в следующем примере:

|  |
| --- |
| unsigned \_\_stdcall thread\_proc(void\*){ // Начало второй транзакции CProxy& prx = CObject::GetObject(CObject::READ\_UNCOMMITTED); // Здесь поток будет ожидать примерно 1 сек. До тех пор, пока // в главном потоке не будет выполнена строчка prx.Commit(); prx.value = 20; prx.Commit(); return 0;}int main(int argc, char\* argv[]){ //Начало транзакции с 0 уровнем изоляции CProxy& prx = CObject::GetObject(CObject::READ\_UNCOMMITTED); //Изменение данных prx.value = 10; //Открываем новую сессию \_beginthreadex(0,0,thread\_proc,0,0,0); //Print CObject::value variable printf("%d\n",prx.value); prx.Commit(); return 0;} |

На экран будет выведено число 10, а второй поток изменит данные только после фиксирования транзакции в главном потоке.

Хорошо, мы избавились от проблемы последнего обновления, но взгляните на следующий код:

|  |
| --- |
| unsigned \_\_stdcall thread\_proc(void\*){ CProxy& prx = CObject::GetObject(CObject::READ\_UNCOMMITTED); prx.value = 20; // Эмулируем работу Sleep(1000); prx.value = 40; prx.Commit(); // Закрытие сессии return 0;}int main(int argc, char\* argv[]){ // Открытие сессии \_beginthreadex(0,0,thread\_proc,0,0,0);  // Эмулируем работу Sleep(100); CProxy& fake = CObject::GetObject(CObject::READ\_UNCOMMITTED); // В этой строчке происходит чтение «грязных данных» // fake.get\_Value() возвращает 20 int\* pAr = new int[fake.get\_Value()]; // Эмулируем работу Sleep(1000); // fake.value = 40 for(int i = 0;i < fake.value;i++) pAr[i] = 0;  if (pAr) delete[] pAr; fake.Commit(); return 0;} |

Если откомпилировать и запустить этот код, он гарантированно приведет к ошибке во время исполнения, так как будет осуществлен выход за границу массива в цикле. Почему? Потому что при создании массива используется значение незафиксированных данных, а в цикле – зафиксированных. Эта проблема известна как проблема «грязного чтения». Она возникает, когда одна транзакция пытается прочитать данные, с которыми работает другая параллельная транзакция. В таком случае временные, неподтвержденные данные могут не удовлетворять ограничениям целостности или правилам. И, хотя к моменту фиксации транзакции они могут быть приведены в «порядок», другая транзакция уже может воспользоваться этими неверными данными, что приведет к нарушению ее работы.

Для решения этой проблемы вводится новый уровень изоляции, на котором запрещается «грязное» чтение. Вот такие изменения нужно внести в реализацию классов CProxy и CObject для того, чтобы программа удовлетворяла второму уровню изоляции:

|  |
| --- |
| class CObject{ friend class CProxy;public: enum {READ\_UNCOMMITTED,READ\_COMMITTED}; static CProxy& GetObject(int level = -1); ~CObject() { DeleteCriticalSection(&exclusive); if (hShared) CloseHandle(hShared); if (hMutex) CloseHandle(hMutex); }protected: CProxy& BeginTran(int level) { return \*(new CProxy(this,level)); } void RequestExclusive(int level) { if (level >= READ\_UNCOMMITTED) TestExclusive(); }  void RequestShared(int level) { if (level > READ\_UNCOMMITTED) TestShared(level); } void RemoveShared(int level) { if (level == READ\_COMMITTED){ RemoveSharedLock();  } } void RemoveLocks() { RemoveAllLocks(); }private: CObject() { value = 0; InitializeCriticalSection(&exclusive); hShared = CreateEvent(NULL,FALSE,TRUE,NULL); } void TestShared(int level) { //Проверка на монопольную блокировку EnterCriticalSection(&exclusive); //Устанавливаем разделяемую блокировку //только если не была установлена монопольная блокировка if (exclusive.RecursionCount == 1) ResetEvent(hShared); //Снимаем монопольную блокировку LeaveCriticalSection(&exclusive); } void TestExclusive() { //Проверка на разделяемую блокировку WaitForSingleObject(hShared,INFINITE); // Проверка на монопольную блокировку EnterCriticalSection(&exclusive); // Вошли больше одного раза if (exclusive.RecursionCount > 1) LeaveCriticalSection(&exclusive); } void RemoveSharedLock() { SetEvent(hShared); } void RemoveAllLocks() { RemoveSharedLock(); // Если была установлена монопольная блокировка - снимаем if (exclusive.OwningThread == (HANDLE)GetCurrentThreadId()) LeaveCriticalSection(&exclusive); } int value; CRITICAL\_SECTION exclusive; HANDLE hShared; static HANDLE hMutex;}; |

Теперь, если изменить константу READ\_UNCOMMITTED в предыдущем примере на READ\_COMMITTED в качестве параметра GetObject, все станет на свои места. При инициализации массива главный поток перейдет в состояние ожидания до тех пор, пока второй поток не выполнит строчку prx.Commit(); Размер массива в главном потоке будет равен 40 элементам.

Хорошо, прекрасно! Где там следующий уровень? :) Чтобы понять, зачем нужен следующий уровень изоляции транзакций «повторяющееся чтение», рассмотрим такой пример:

|  |
| --- |
| unsigned \_\_stdcall thread\_proc(void\*){ { // Начало транзакции CProxy& prx = CObject::GetObject(CObject::READ\_COMMITTED); prx.value = 20; prx.Commit(); } // Эмулируем работу Sleep(500); { // Начало транзакции CProxy& prx = CObject::GetObject(CObject::READ\_COMMITTED); prx.value = 40; prx.Commit(); } return 0;}int main(int argc, char\* argv[]){ // Начало сессии \_beginthreadex(0,0,thread\_proc,0,0,0);  // Эмулируем работу Sleep(100); CProxy& fake = CObject::GetObject(CObject::READ\_COMMITTED); // Создание массива int\* pAr = new int[fake.get\_Value()];  // Эмулируем работу Sleep(1000); // Инициализация массива for(int i = 0;i < fake.value;i++) pAr[i] = 0; if (pAr) delete[] pAr; fake.Commit(); return 0;} |

Если запустить этот пример, он, как и предыдущий, приведет к ошибке доступа к памяти. Дело в том, что изначально создается массив размером в 20 элементов, а в цикле инициализации используется значение 40, и на 21 элементе мы получим ошибку доступа.

Проблема повторного чтения состоит в том, что между операциями чтения в одной транзакции другие транзакции могут беспрепятственно вносить любые изменения, так что повторное чтение тех же данные приведет к другому результату.

Для поддержки третьего уровня изоляции в код изменений вносить не надо! :) Необходимо лишь не снимать разделяемые блокировки до конца транзакции. Так как метод, приведенный ниже, снимает блокировку только на уровне READ\_COMMITTED:

|  |
| --- |
|  void RemoveShared(int level) { if (level == READ\_COMMITTED){ RemoveSharedLock();  } } |

нам нужно лишь добавить новую константу в перечисление типов блокировок.

|  |
| --- |
|  enum {READ\_UNCOMMITTED,READ\_COMMITTED,REPEATABLE\_READ}; |

Теперь, если в приведенном выше примере изменить константу READ\_COMMITTED на REPEATABLE\_READ в качестве параметра GetObject, код заработает правильно и без ошибок.

|  |
| --- |
| ПРИМЕЧАНИЕСовершенно не обязательно менять уровень изоляции транзакций в потоке thread\_proc, работа примера не изменится, даже если изменить уровень изоляции на READ\_UNCOMMITTED. |

Здесь мы ставим блокировку обновления, если транзакция читает данные с уровнем изоляции REPEATABLE\_READ.

В заключение, перед тем как привести полностью код с поддержкой первых трех уровней изоляции, давайте поговорим вот о чем. Созданный код реализует блокирующую модель, которая характерна для СУБД MS SQL Server 2000. Существует также версионная модель реализации блокировок, которую поддерживает такая известная СУБД, как Oracle. Чем отличаются эти модели? Рассмотрим такой код:

|  |
| --- |
| unsigned \_\_stdcall thread\_proc(void\*){ // Print CObject::value variable CProxy& fake = CObject::GetObject(); printf("in second session: %d\n",fake.value); fake.Commit(); return 0;}int main(int argc, char\* argv[]){ // Начало транзакции CProxy& prx = CObject::GetObject(); prx.value = 10; // Начало новой сессии \_beginthreadex(0,0,thread\_proc,0,0,0); // Эмулируем работу Sleep(100); printf("in primary session: %d\n",prx.value); prx.Commit(); return 0;} |

Здесь во второй сессии (выполняемой в отдельном потоке) мы просто читаем данные и выводим их на консоль. Так как значение переменной value мы изменили перед стартом второй сессии, совершенно очевидно, что на экран будет выведено

|  |
| --- |
| in second session: 10in primary session: 10 |

Однако при использовании версионной модели мы должны получить

|  |
| --- |
| in second session: 0in primary session: 10 |

Причина в том, что для каждой транзакции хранится своя копия данных (snap-shot), которая синхронизируется с основными данными только в момент фиксирования транзакции.

|  |
| --- |
| ПРИМЕЧАНИЕOracle хранит эти копии данных в специальном хранилище, который называется rollback segment. |

Версионная модель характеризуется тем, что в ней отсутствует нулевой уровень изоляции транзакций (READ UNCOMMITTED), и вместо него вводится новый уровень, который в приведенном далее коде я назвал SNAP\_SHOT. Он отличается от стандартного тем, что позволяет читать действительные зафиксированные данные, даже при наличии незавершенных транзакций обновления.

Вот конечный вариант классов CProxy и CObject, который реализует обе модели и, вдобавок к этому, поддерживает два «хинта»: UPDLOCK и XLOCK. Они предназначены для изменения уровня изоляции непосредственно при работе со значением переменной, а их смысл я поясню в следующих разделах.

|  |
| --- |
| #define MSSQL// #define ORACLEclass CObject;class CProxy{ friend class CObject;public: \_\_declspec(property(get=get\_Value,put=put\_Value)) int value; int get\_Value(int level = -1) const; void put\_Value(int i); void Commit(); void Rollback();private:  int \_level; int \_value; bool fUpd; CProxy(CObject\* par,int level) { fUpd = false; parent = par; \_level = level; } CObject\* parent;};class CObject{ friend class CProxy;public: enum {#ifdef MSSQL READ\_UNCOMMITTED,#elif defined ORACLE SNAP\_SHOT,#endif READ\_COMMITTED,REPEATABLE\_READ,UPDLOCK,XLOCK}; static CProxy& GetObject(int level = -1); ~CObject() { DeleteCriticalSection(&exclusive); DeleteCriticalSection(&update); if (hShared) CloseHandle(hShared); if (hMutex) CloseHandle(hMutex); }protected: CProxy& BeginTran(int level) { return \*(new CProxy(this,level)); } void RequestExclusive(int level) { ATLASSERT(level <= REPEATABLE\_READ);#ifdef MSSQL if (level >= READ\_UNCOMMITTED)#elif defined ORACLE if (level >= SNAP\_SHOT)#endif TestExclusive(); }  void RequestShared(int level) {#ifdef MSSQL if (level > READ\_UNCOMMITTED)#elif defined ORACLE if (level > SNAP\_SHOT)#endif TestShared(level); } void RemoveShared(int level) { if (level == READ\_COMMITTED){ RemoveSharedLock();  } } void RemoveLocks() { RemoveAllLocks(); }private: CObject() { value = 0; InitializeCriticalSection(&update); InitializeCriticalSection(&exclusive); hShared = CreateEvent(NULL,FALSE,TRUE,NULL); } void TestShared(int level) { // Проверка на монопольную блокировку EnterCriticalSection(&exclusive); // Устанавливаем блокировку обновления if (level == UPDLOCK){ EnterCriticalSection(&update); // Вошли больше одного раза if (update.RecursionCount > 1) LeaveCriticalSection(&update); } else if (level != XLOCK){  // Устанавливаем разделяемую блокировку // только если не была установлена блокировка обновления или // монопольная блокировка if (update.OwningThread != (HANDLE)GetCurrentThreadId() && exclusive.RecursionCount == 1) ResetEvent(hShared); // Снимаем монопольную блокировку LeaveCriticalSection(&exclusive); } // Если указан XLOCK монопольная блокировка остается } void TestExclusive() { // Проверка на разделяемую блокировку WaitForSingleObject(hShared,INFINITE); // Проверка на блокировку обновления EnterCriticalSection(&update); // Проверка на монопольную блокировку EnterCriticalSection(&exclusive); // Снимаем блокировку обновления LeaveCriticalSection(&update); // Вошли больше одного раза if (exclusive.RecursionCount > 1) LeaveCriticalSection(&exclusive); } void RemoveSharedLock() { SetEvent(hShared); } void RemoveAllLocks() { RemoveSharedLock();  // Если была установлена блокировка обновления - снимаем if (update.OwningThread == (HANDLE)GetCurrentThreadId()) LeaveCriticalSection(&update); // Если была установлена монопольная блокировка - снимаем if (exclusive.OwningThread == (HANDLE)GetCurrentThreadId()) LeaveCriticalSection(&exclusive); } int value; CRITICAL\_SECTION update; CRITICAL\_SECTION exclusive; HANDLE hShared; static HANDLE hMutex;};\_\_declspec(selectany) HANDLE CObject::hMutex = NULL;CProxy& CObject::GetObject(int level){ HANDLE hLocMutex = CreateMutex(NULL,TRUE,\_T("Guard-Lock-Mutex")); bool flg = GetLastError() == ERROR\_ALREADY\_EXISTS;  if (flg) WaitForSingleObject(hLocMutex,INFINITE); else CObject::hMutex = hLocMutex; static CObject obj; ReleaseMutex(hLocMutex);  if (flg) CloseHandle(hLocMutex); return obj.BeginTran(level);}void CProxy::Commit(){#ifdef ORACLE parent->value = \_value;#endif parent->RemoveLocks(); delete this;}void CProxy::Rollback(){#ifdef MSSQL if (fUpd) parent->value = \_value;#endif parent->RemoveLocks(); delete this;}void CProxy::put\_Value(int i){ parent->RequestExclusive(\_level);#ifdef MSSQL if (!fUpd) \_value = parent->value; parent->value = i;#elif defined ORACLE \_value = i;#endif fUpd = true;}int CProxy::get\_Value(int level) const{ if (level == -1)  level = \_level; parent->RequestShared(level);#ifdef MSSQL int v = parent->value; parent->RemoveShared(level); return v;#elif defined ORACLE return \_value;#endif} |

Из этих примеров должно быть понятно, что блокировки – дело серьезное. :) Но, прежде чем перейти к рассмотрению их реализации в MS SQL Server 2000, я приведу обещанные в начале уровни определения изоляции транзакций. Каждый уровень включает в себя предыдущий с предъявлением более жестких требований к изоляции.

No trashing of data (запрещение «загрязнения» данных). Запрещается изменение одних их тех же данных двумя и более параллельными транзакциями. Изменять данные может только одна транзакция, если какая-то другая транзакция попытается сделать это, она должна быть заблокирована до окончания работы первой транзакции.

No dirty read (запрещение «грязного» чтения). Если данная транзакция изменяет данные, другим транзакциям запрещается читать эти данные до тех пор, пока первая транзакция не завершится.

No nonrepeatable read (запрещение неповторяемого чтения). Если данная транзакция читает данные, запрещается изменять эти данные до тех пор, пока первая транзакция не завершит работу. При этом другие транзакции могут получать доступ на чтение данных.

No phantom (запрещение фантомов). Если данная транзакция производит выборку данных, соответствующих какому-либо логическому условию, другие транзакции не могут ни изменять эти данные, ни вставлять новые данные, которые удовлетворяют тому же логическому условию.

Если вы не совсем поняли суть последнего уровня изоляции, не расстраивайтесь. Я специально оставил его на потом, так как в данный момент нет возможности рассмотреть примеры, описывающие проблему и механизмы ее избежания.

В таблице 1 подводится итог этого раздела и изучения уровней изоляции.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Уровни изоляции | Загрязнение данных | Грязное чтение | Неповторяемое чтение | Фантомы |
| READ UNCOMMITTED | - | + | + | + |
| READ COMMITTED | - | - | + | + |
| REPEATABLE READ | - | - | - | + |
| SERIALIZABLE | - | - | - | - |

Блокировки

Блокировки в MS SQL Server 2000 (в дальнейшем просто сервер) – это механизм реализации требования изолированности транзакций. Вся последующая информация специфична только для указанного сервера.

Существует три основных типа блокировок и множество специфичных. Сервер устанавливает блокировки автоматически в зависимости от текущего уровня изоляции транзакции, однако при желании вы можете изменить тип с помощью специальных подсказок – хинтов.

При открытии новой сессии по умолчанию выбирается уровень изоляции READ COMMITTED. Вы можете изменить этот уровень для данного соединения с помощью команды:

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL |

Более подробно эту команду и хинты для операторов T-SQL мы рассмотрим в следующем разделе. Пока же я хочу подробно остановиться на типах блокировок.

Блокировки применяются для защиты совместно используемых ресурсов сервера. В качестве объектов блокировок могут выступать следующие сущности:

База данных (обозначается DB). При наложении блокировки на базу данных блокируются все входящие в нее таблицы.

Таблица (обозначается TAB). При наложении блокировки на таблицу блокируются все экстенты данной таблицы, а также все ее индексы.

|  |
| --- |
| ПРИМЕЧАНИЕЭкстент – это группа из 8 страниц.Страница – минимальная единица хранения данных в файле базы данных. Размер страницы составляет 8 Кб. |

Экстент (обозначается EXT). При наложении блокировки на экстент блокируются все страницы, входящие в данный экстент.

Страница (обозначается PAG). При наложении блокировки на страницу блокируются все строки данной страницы.

Строка (обозначается RID).

Диапазон индекса (обозначается KEY). Блокируются данные, соответствующие диапазону индекса, на обновление, вставку и удаление.

SQL Server сам выбирает наиболее оптимальный объект для блокировки, однако пользователь может изменить это поведение с помощью тех же хинтов. При автоматическом определении объекта блокировки сервер должен выбрать наиболее подходящий с точки зрения производительности и параллельной работы пользователей. Чем меньше детализация блокировки (строка – самая высокая степень детализации), тем ниже ее стоимость, но ниже и возможность параллельной работы пользователей. Если выбирать минимальную степень детализации, запросы на выборку и обновление данных будут исполняться очень быстро, но другие пользователи при этом должны будут ожидать завершения транзакции. Степень параллелизма можно увеличить путем повышения уровня детализации, однако блокировка – вполне конкретный ресурс SQL Server’а, для ее создания, поддержания и удаления требуется время и память.

|  |
| --- |
| ПРИМЕЧАНИЕБлокировка занимает 96 байт. [1] Общее количество блокировок может варьироваться от 5000 до 2 147 483 647. Конкретное значение можно задать с помощью хранимой процедуры sp\_configure с параметром locks. |

SQL Server может принимать решение об уменьшении степени детализации, когда количество блокированных ресурсов увеличивается. Этот процесс называется эскалацией блокировок.

Вообще говоря, существует два метода управления конкуренцией для обеспечения параллельной работы множества пользователей – оптимистический и пессимистический. SQL Server использует оптимистическую конкуренцию только при использовании курсоров (cursors). Для обычных запросов на выборку и обновление используется пессимистическая конкуренция. Рассмотрим подробнее, что они собой представляют:

Оптимистический метод управления характеризуется тем, что вместо непосредственного чтения данных берется значение из буфере. Никаких блокировок при этом не накладывается. Другие транзакции могут спокойно читать или даже изменять данные. В момент фиксирования транзакции система сравнивает предыдущее (заранее сохраненное) значение данных с текущим. Если они совпадают, выполняются операции блокировки, обновления и разблокировки данных. Если же значения отличаются, то система генерирует ошибку и откатывает транзакцию. Хотя такой подход не удовлетворяет требованиям стандарта, он позволяет в определенных случаях добиться лучшей производительности, чем пессимистический подход. Достоинства этого режима очевидны: система не теряет времени на установку блокировок и ресурсов для их создания. Однако для систем с большим количеством пользователей, часто изменяющих данные, такой режим использовать не рекомендуется, так как цена отката транзакции и ее повторного выполнения значительно выше установки блокировки при чтении данных.

Пессимистический метод. В этом случае сервер всегда блокирует ресурсы в соответствии с текущим уровнем изоляции. В примере предыдущего раздела использовался именно этот метод управления конкуренцией, однако совсем не сложно адаптировать его для поддержки оптимистического режима.

Блокировки – чрезвычайно важный и неотъемлемый механизм функционирования сервера. Они применяются для каждого запроса на чтение или обновления данных, а также во многих других случаях (например, при создании новой сессии). Работой с блокировками занимается специальный модуль SQL Server’а – менеджер блокировок (Lock Manager). В его задачи входит:

создание и установка блокировок;

снятие блокировок;

эскалация блокировок;

определение совместимости блокировок;

устранение взаимоблокировок (deadlocks) и многое другое.

Когда пользователь делает запрос на обновление или чтение данных, менеджер транзакций передает управление менеджеру блокировок для того, чтобы выяснить были ли блокированы запрашиваемые ресурсы, и, если да, совместима ли запрашиваемая блокировка с текущей. Если блокировки несовместимы, выполнение текущей транзакции откладывается до тех пор, пока данные не будут разблокированы. Как только данные становятся доступны, менеджер блокировок накладывает запрашиваемую блокировку, и возвращает управление менеджеру транзакций.

Момент снятия блокировки сильно зависит от текущего уровня изоляции и типа запроса. Например, при выполнении запроса на выборку данных при уровне изоляции ниже REPEATABLE READ менеджер блокировок снимает блокировку сразу же после извлечения всех данных с текущей страницы. При этом происходит установка блокировки на следующую страницу.

Простые блокировки

SQL Server поддерживает три основных типа блокировок:

Разделяемая блокировка (Shared Lock), обозначается латинской буквой S. Эта самый распространенный тип блокировки, который используется при выполнении операции чтения данных. Гарантируется что данные, на которые она наложена, не будут изменены другой транзакцией. Однако чтение данных возможно.

Монопольная блокировка (Exclusive Lock), обозначается латинской буквой X. Этот тип применяется при изменении данных. Если на ресурс установлена монопольная блокировка, гарантируется, что другие транзакции не могут не только изменять данные, но даже читать их.

Блокировка обновления (Update Lock), обозначается латинской буквой U. Эта блокировка является промежуточной между разделяемой и монопольной блокировкой. Так как монопольная блокировка не совместима ни с одним видом других блокировок (есть одно исключение, о котором попозже), ее установка приводит к полному блокированию ресурса. Если транзакция хочет обновить данные в какой-то ближайший момент времени, но не сейчас, и, когда этот момент придет, не хочет ожидать другой транзакции, она может запросить блокировку обновления. В этом случае другим транзакциям разрешается устанавливать разделяемые блокировки, но не позволяет устанавливать монопольные. Другими словами, если данная транзакция установила на ресурс блокировку обновления, никакая другая транзакция не сможет получить на этот же ресурс монопольную блокировку или блокировку обновления до тех пор, пока установившая блокировку транзакция не будет завершена.

Прежде чем идти дальше, давайте рассмотрим небольшой пример. Для просмотра текущих блокировок существует системная хранимая функция sp\_lock. Она возвращает информацию о блокировках в формате, описанном в таблице 2.

|  |  |
| --- | --- |
| Имя колонки | Описание |
| spid | Идентификатор процесса SQL Server. |
| dbid | Идентификатор базы данных. |
| ObjId | Идентификатор объекта, на который установлена блокировка. |
| IndId | Идентификатор индекса. |
| Type | Тип объекта. Может принимать значения: DB, EXT, TAB, PAG, RID, KEY. |
| Resource | Содержимое колонки syslocksinfo.restext. Обычно это идентификатор строки (для типа RID) или идентификатор страницы (для типа PAG). |
| Mode | Тип блокировки. Может принимать значения: Sch-S, Sch-M, S, U, X, IS, IU, IX, SIU, SIX, UIX, BU, RangeS-S, RangeS-U, RangeIn-Null, RangeIn-S, RangeIn-U, RangeIn-X, RangeX-S, RangeX-U, RangeX-X. Об этих значениях будет сказано ниже. |
| Status | Статус процесса SQL Server. Может принимать значения: GRANT, WAIT, CNVRT. |

Эта процедура возвращает данные о блокировках из системной таблицы syslockinfo, которая находится в базе данных master.

|  |
| --- |
| ПРИМЕЧАНИЕИнформация именно из этой таблицы используется менеджером блокировок для определения совместимости блокировок при запросе ресурсов транзакциями. |

Во всех примерах используется таблица test, которая создается следующим скриптом:

|  |
| --- |
| create table test(i int, n varchar(20))insert into test values(1,’alex’)insert into test values(2,’rosa’)insert into test values(3,’dima’) |

Во-первых, давайте действительно убедимся, что при чтении данных с уровнем изоляции ниже REPEATABLE READ разделяемые блокировки снимаются сразу же после извлечения данных:

|  |
| --- |
| print @@spidbegin tran select \* from test |

Мы начали транзакцию, но оставили ее открытой. Для того чтобы посмотреть, какие блокировки наложены предыдущим скриптом, вызовем процедуру sp\_lock (в другой сессии) с параметром, выведенным print @@spid (у меня это 54).

|  |
| --- |
| СОВЕТТекущее значение идентификатора процесса сервера можно увидеть в строке состояния программы Query Analizer. |

|  |
| --- |
| sp\_lock 54 |

Результат приведен в таблице 3.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| spdi | dbid | ObjId | IndId | Type | Resource | Mode | Status |
| 54 | 8 | 0 | 0 | DB |  | S | GRANT |

Мы видим стандартную блокировку, которая создается для каждого соединения с базой данных. Никакой дополнительной блокировки установлено не было.

Другим способом проверки того, что блокировка снимается сразу после извлечения данных, является использование трассировки. Попробуйте выполнить следующую команду:

|  |
| --- |
| dbcc traceon(3604,1200)select \* from testdbcc traceoff(3604,1200) |

Флаг трассировки 3604 заставляет сервер передавать отладочную информацию в текущее соединение непосредственно клиенту, а 1200 – выводить информацию о блокировках. В результате мы получим следующее:

|  |
| --- |
| Process 54 acquiring S lock on DB: 8 (class bit0 ref1) result: OKProcess 54 acquiring IS lock on TAB: 8:1993058136 [] (class bit0 ref1) result: OKProcess 54 acquiring IS lock on PAG: 8:1:31 (class bit0 ref1) result: OKi n ----------- -------------------- 1 alex2 rosa3 dima(3 row(s) affected)Process 54 releasing lock on PAG: 8:1:31Process 54 releasing lock on TAB: 8:1993058136 [] |

Хорошо. В первой сессии зафиксируйте транзакцию:

|  |
| --- |
| --print @@spid--begin tran select \* from testcommit |

Повторный вызов sp\_lock приводит к тем же результатам. Это подтверждает, что предыдущим запросом никаких блокировок не устанавливалось. Теперь попробуем наложить блокировку обновления. Делается это с помощью хинта updlock (хинты подробно будут рассмотрены далее):

|  |
| --- |
| begin tran select \* from test with (updlock) |

Теперь вызов sp\_lock 54 дает более интересный результат (таблица 4):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| spdi | dbid | ObjId | IndId | Type | Resource | Mode | Status |
| 54 | 8 | 0 | 0 | DB |  | S | GRANT |
| 54 | 8 | 1993058136 | 0 | RID | 1:29:2  | U | GRANT |
| 54 | 8 | 1993058136 | 0 | RID | 1:29:0 | U | GRANT |
| 54 | 8 | 1993058136 | 0 | PAG | 1:29 | IU | GRANT |
| 54 | 8 | 1993058136 | 0 | TAB |  | IX | GRANT |
| 54 | 8 | 1993058136 | 0 | RID | 1:29:1 | U | GRANT |

Как видно, на три строки была наложена блокировка обновления, что означает невозможность обновления этих строк другими транзакциями. Кроме этого, были наложены еще две блокировки, которые относятся к типу блокировок намерения (intent locks) – блокировка на страницу и на таблицу. Пусть вас не смущает колонка Resource – ее содержание я опишу позже. Поле ObjId представляет собой идентификатор объекта, имя которого можно получить с помощью функции object\_name. IndId – идентификатор индекса, в данном примере всегда равен 0, так как в таблице test индексов не используется.

Блокировки намерений

Блокировки намерений всегда устанавливаются на таблицу или страницу, но никогда – на строку. Для чего они нужны? Блокировки намерений относятся к специальным типам блокировок и предназначены для повышения производительности работы менеджера блокировок.

Предположим, некая транзакция пытается изменить какую-либо строку в таблице test. Чтобы определить, что эту транзакцию необходимо заблокировать, менеджеру транзакций (в отсутствие блокировок намерения) пришлось бы сканировать всю таблицу syslockinfo для проверки всех строк таблицы test. Чтобы избежать этой неблагодарной работы, менеджер блокировок сразу устанавливает на страницу и таблицу блокировку намерения обновления (Intent Update) и монопольную блокировку намерения (Intent Exclusive) соответственно, и проверяет уже только их. Таким образом, блокировки намерений – это хинт менеджеру блокировок, который дает значительный выигрыш в производительности при проверке совместимости блокировок.

Рассмотрим блокировки намерений более подробно:

Разделяемая блокировка намерений (обозначается IS). Этот тип блокировки предназначен для того, чтобы вместо проверки множества разделяемых блокировок для каждой строки в момент обновления данных, или установки множества разделяемых блокировок для каждой строки в момент чтения данных, можно было установить одну блокировку намерений для страницы или всей таблицы. Этот тип блокировки совместим со всеми остальными режимами, кроме монопольной блокировки. Другими словами, если транзакция пытается обновить данные, на которые наложена разделяемая блокировка намерений, ее выполнение заблокируется до момента завершения первой транзакции.

Монопольная блокировка намерений (обозначается IX). Монопольная блокировка намерений запрещает доступ к данным какого-либо вида (даже на чтение) и позволяет повысить эффективность работы при множественном обновлении данных. В отличие от простой монопольной блокировки она совместима сама с собой. Т.е. другая транзакция также может наложить монопольную блокировку намерений на те же ресурсы, что и первая транзакция.

Разделяемо-монопольная блокировка намерений (обозначается SIX). Подсказывает менеджеру блокировок, что транзакция в основном выполняет чтение данных и лишь частично, в очень небольшом объеме – их изменение. Основное отличие данного типа блокировки от монопольной блокировки намерений состоит в том, что она совместима с разделяемой блокировкой намерений и не совместима ни с какой другой, включая саму себя.

Чтобы уяснить работу этого типа блокировок, продолжим наш пример. Создайте новую сессию (у меня ее идентификатор равен 55) и выполните следующий скрипт:

|  |
| --- |
| begin tran insert into test values(4,'other') |

На удивление многих она прекрасно отработает и не выдаст никаких предупреждений. Как же так? Посмотрим, что скажет нам sp\_lock

|  |
| --- |
| sp\_lock 54,55 |

Результат приведен в таблице 5:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| spid | dbid | ObjId | IndId | Type | Resource | Mode | Status |
| 54 | 8 | 0 | 0 | DB |  | S | GRANT |
| 54 | 8 | 1993058136 | 0 | RID | 1:29:02 | U | GRANT |
| 54 | 8 | 1993058136 | 0 | RID | 1:29:00 | U | GRANT |
| 54 | 8 | 1993058136 | 0 | PAG | 1:29 | IU | GRANT |
| 54 | 8 | 1993058136 | 0 | TAB |  | IX | GRANT |
| 54 | 8 | 1993058136 | 0 | RID | 1:29:01 | U | GRANT |
| 55 | 8 | 0 | 0 | DB |  | S | GRANT |
| 55 | 8 | 1993058136 | 0 | PAG | 1:29 | IX | GRANT |
| 55 | 8 | 1993058136 | 0 | TAB |  | IX | GRANT |
| 55 | 8 | 1993058136 | 0 | RID | 1:29:03 | X | GRANT |

Как видно, предыдущие блокировки остались (еще бы! мы же не зафиксировали транзакцию), и добавились четыре новых: одна блокировка базы, не имеющая никакого значения, две блокировки намерений (на таблицу и страницу) и монопольная блокировка на новую строку (идентификатор 1:29:03).

Пока все более-менее ясно. Однако если выполнить в первой сессии обновление данных, ради которых, собственно, и была наложена блокировка обновления, ничего не выйдет! При выполнении следующей команды сессия заблокируется:

|  |
| --- |
| --begin tran select \* from test with (updlock)update test set n = 'other1' where i = 1 |

Если посмотреть на содержимое таблицы syslockinfo с помощью хранимой процедуры sp\_lock, можно увидеть, что добавилась еще одна строка (таблица 6).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| spid | dbid | ObjId | IndId | Type | Resource | Mode | Status |
| 54 | 8 | 1993058136 | 0 | RID | 1:29:03 | U | WAIT |

Так как на строку 1:29:03 уже наложена монопольная блокировка, никакие другие блокировки на этот же ресурс наложены быть не могут. Но каким образом наш безобидный update на первую строчку привел к наложению блокировки обновления на четвертую строчку? Дело в том, что при отсутствии индекса менеджер блокировок не может определить, какой именно строке соответствует заданное логическое условие, поэтому производится сканирование и установка монопольной блокировки на каждую строчку таблицы. При наличии индекса SQL Server использует другой метод, основанный на блокировках диапазона индекса. Чтобы перейти к их рассмотрению, вернитесь в сессию с идентификатором 55 и выполните команду отката транзакции:

|  |
| --- |
| --begin tran insert into test values(4,'other')rollback |

После этого нужно откатиться и в первой сессии:

|  |
| --- |
| --begin tran select \* from test with (updlock)--update test set n = 'other1' where i = 1rollback |

Блокировка диапазона ключа или индекса позволяет заблокировать данные на основе логического значения, такого, как выражение в разделе where. Кроме всего прочего, только этот тип блокировок позволяет эффективно решить проблему чтения фантомов. Эта проблема связана с тем, что даже при наличии уровня изоляции REPEATABLE READ вы можете получить разное количество строк, соответствующих определенному логическому условию в течение одной транзакции (кстати, логическое условие может и отсутствовать). Это может произойти из-за того, что не существует никаких ограничений на вставку новых строк, несмотря на то, что обновление выбранных строк заблокировано разделяемой блокировкой. Новые строки могут содержать значения, удовлетворяющие логическому запросу в разделе where, так что при следующем запросе количество строк в результирующей выборке изменится.

SQL Server поддерживает уровень SERIALIZABLE и для таблиц, не имеющих индекса. Однако в этом случае на всю таблицу целиком накладывается разделяемая блокировка до конца транзакции, что очень сильно снижает эффективность параллельной работы пользователей.

|  |
| --- |
| ПРИМЕЧАНИЕДо выхода MS SQL Server 7.0 менеджер блокировок поступал таким образом (блокировал всю таблицу) с любыми таблицами, работа с которыми велась с уровнем изоляции SERIALIZABLE. |

Блокировки записи и диапазона индекса

Существует два вида блокировки индекса: блокировка самой записи индекса (index entry) и блокировка диапазона. Блокировка записи индекса эквивалентна блокировке строки, т.е. менеджер блокировок будет блокировать строку, если в таблице отсутствует индекс, или же будет блокировать запись индекса, которая соответствует данной строке. Поскольку индекс может быть построен по неуникальному полю, блокируемая запись не может быть идентифицирована только записью индекса, поэтому для вычисления локатора ресурса применяется алгоритм хеширования идентификатора строки (RID) и записи индекса.

Если бы в нашей таблице существовал индекс по полю i, то в предыдущей таблице блокировок в колонке Resource фигурировали бы хеш-значения записей индекса, а тип ресурса был – KEY.

Блокировка диапазона ключа существенно отличается от всего сказанного выше. Она позволяет наложить блокировку не на конкретные строки или записи индекса, а на диапазон строк, что позволяет блокировать записи по некоторому логическому условию. Естественно, что SQL Server не понимает логических выражений в том виде, каком понимает их автор запроса, и не может абсолютно точно наложить требуемую блокировку. Рассмотрим теоретические аспекты реализации блокировки по логическому условию.

Допустим, у нас есть диапазон значений от A до B. Если мы хотим прочитать какое-либо значение С, которое находится между A и B, но не равно им, возможны следующие варианты:

Если С не существует, накладывается разделяемая блокировка на весь диапазон (A,B), при этом вы не можете вставить никакое значение в этот диапазон. Круглая скобка означает, что значение не входит в диапазон. Кроме этого, вы не можете изменить значения A и B на С. Объяснение: Если позволить вставлять записи в диапазон (A,B) или изменять значения A и B на С, результаты повторного запроса могут измениться, что приведет к появлению фантома. Менеджер блокировок SQL Server’а, по техническим причинам, которые будут рассмотрены далее, накладывает несколько отличные блокировки: запрещается вставка в диапазон [A,B), т.е. запрещается вставлять значения, равные А, и запрещается изменять значение B не только на С, но и на любое другое.

Если С существует, накладывается разделяемая блокировка на диапазоны (A,C] и [C,B). При этом запрещается вставлять значения из диапазона (A,B) и изменять A и B на С. Менеджер блокировок SQL Server’а по техническим причинам накладывает отличные блокировки: запрещается вставка в диапазон [A,B), т.е. запрещается вставлять значения, равные А, и запрещается изменять значение B не только на С, но и на любое другое.

Если мы хотим вставить значение С, которое находится между A и B, то:

Накладывается разделяемая блокировка на диапазон [A, B]. Менеджер блокировок вместо разделяемой блокировки накладывает специальный тип блокировки, предназначенный только для тестирования диапазона индекса.

Вставляется значение С, разделяемая блокировка [A, B] снимается.

Накладывается монопольная блокировка на ключ C.

При удалении значения С, которое находится между A и B, возможны следующие ситуации:

Если значение С отсутствует, накладывается блокировка обновления диапазона на (A,B). Менеджер блокировок SQL Server’а по техническим причинам накладывает блокировку обновления на диапазон (A,B], т.е. вы можете изменить значение A, но не значение B.

Если значение С существует, накладывается разделяемая блокировка на диапазон [A,B]. Затем накладывается монопольная блокировка на ключ C. SQL Server производит несколько отличные действия, а именно: он сначала накладывает монопольную блокировку на диапазон [A,C], а затем блокировку обновления на диапазон [C,B].

При блокировании диапазона индекса SQL Server накладывает блокировку как на саму запись индекса, так и на диапазон значений от текущего значения ключа до ближайшего следующего. Диапазон может блокироваться только двумя способами: разделяемая блокировка и монопольная блокировка. Кроме этого, есть специальный тип блокировки диапазона, который применяется только для тестирования наличия других блокировок индекса и, соответственно, не совместим с ними. Не бывает блокировок диапазона без блокировки записи индекса, за исключением блокировки тестирования. Блокировка диапазона и блокировка записи индекса объединяются, формируя, таким образом, новые типы блокировок. Рассмотрим их (таблица 7).

|  |  |  |  |
| --- | --- | --- | --- |
| Диапазон | Запись индекса | Тип | Описание |
| RangeS | S | RangeS\_S | Разделяемая блокировка диапазона, разделяемая блокировка записи индекса. |
| RangeS | U | RangeS\_U | Разделяемая блокировка диапазона, блокировка обновления записи индекса. |
| RangeX | X | RangeX\_X | Монопольная блокировка на диапазон и запись индекса. |
| RangeI | нет | RangeI\_N | Блокировка тестирования. Применяется перед вставкой нового значения в диапазон. |

Блокировка диапазона накладывается на диапазон значений от текущего ключа до ближайшего следующего. Если следующего ключа не существует, блокировка накладывается на все значения, начиная с текущего. Так как один диапазон соответствует двум записям индекса, количество диапазонов должно быть меньше количества выбранных (или изменяемых) строк на 1. Но для того, чтобы защитить граничные значения записей индекса (например, от вставки такого же значения) накладывается еще две дополнительные блокировки индекса, расширяющие общий диапазон справа и слева. Таким образом, при выборе N’ого количества строк, количество блокировок диапазона будет всегда равно N+1.

В таблице 8 приведена матрица совместимости блокировок диапазона индекса.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип | RangeS\_S | RangeS\_U | RangeX\_X | RangeI\_N |
| RangeS\_S | + | + | - | - |
| RangeS\_U | + | - | - | - |
| RangeX\_X | - | - | - | + |
| RangeI\_N | - | - | + | - |

|  |
| --- |
| ПРИМЕЧАНИЕБлокировка RangeI\_N совместима с любым простым типом блокировки (даже с монопольной блокировкой) исключая Sch-M. |

Давайте рассмотрим несколько примеров, использующих блокировки диапазона. Для этого потребуется новая таблица, создать которую можно следующим образом:

|  |
| --- |
| create table test\_key(i int)insert into test\_key values(3)insert into test\_key values(5)insert into test\_key values(7)insert into test\_key values(9)create nonclustered index \_i\_test\_key on test\_key(i) |

Чтобы не напрягаться и не писать каждый раз begin tran, все соединения переключим в режим неявной транзакции, при котором SQL Server не выполняет автоматического фиксирования транзакции после успешного выполнения каждой команды. Новая транзакция начинается сразу же после ручного фиксирования текущей. Режимы переключаются с помощью следующей команды:

|  |
| --- |
| SET IMPLICIT\_TRANSACTIONS { ON | OFF } |

Текущее состояние этой переменной можно определить следующим способом:

|  |
| --- |
| select case when (@@options & 2) = 2 then 'IMPLICIT' else 'NOT IMPLICIT' end |

В дальнейшем все примеры, если явно не оговорено другое, выполняются в режиме неявной транзакции.

Для начала посмотрим, что такое разделяемая блокировка диапазона:

|  |
| --- |
| set implicit\_transactions onselect \* from test\_key with (serializable) |

Если сейчас заглянуть в табличку syslockinfo или вызвать хранимую процедуру sp\_lock, можно увидеть 5 блокировок типа RangeS\_S. Значение в строке Resource однозначно идентифицирует запись индекса, однако для одной строки значение равно (ffffffffffff). Это говорит о том, что весь диапазон чисел от -INF до +INF (где INF - бесконечности) заблокирован. Если вы попытаетесь вставить в таблицу test\_key значение, например, 68, у вас ничего не выйдет. Рассмотрим результаты вызова процедуры sp\_lock (приведены сокращенно в таблице 9):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| spid | dbid | ObjId | IndId | Type | Resource | Mode | Status |
| 55 | 8 | 2009058193 | 2 | KEY | (ffffffffffff) | RangeS-S | GRANT |
| 56 | 8 | 2009058193 | 0 | RID | 1:29:04 | X | GRANT |
| 56 | 8 | 0 | 0 | PAG | 1:29 | IX | GRANT |
| 56 | 8 | 2009058193 | 2 | PAG | 1:34 | IX | GRANT |
| 56 | 8 | 2009058193 | 2 | KEY | (ffffffffffff) | RangeIn | WAIT |
| 56 | 8 | 2009058193 | 0 | TAB |  | IX | GRANT |

Так как тип блокировки RangeIn (aka RangeI) несовместим ни с каким другим типом блокировки диапазона, сессия за номером 56 блокируется. Примечательный факт – само значение все-таки было вставлено в таблицу, о чем свидетельствует выделенная строка, и только в момент обновления индекса возникла проблема совместимости блокировок. Чтобы убедится, что значение действительно было вставлено, можно воспользоваться командой dbcc page, которая подробно описана в следующем разделе. Кроме этого видно, что установлены блокировки намерений на таблицу и две страницы: страницу с данными и страницу, на которой располагается индекс.

|  |
| --- |
| ПРИМЕЧАНИЕИндекс располагается на другой странице вследствие того, что он является некластерным. |

Теперь отменим транзакцию и выполним такой код:

|  |
| --- |
| --set implicit\_transactions on--select \* from test\_key with (serializable)delete from test\_key with (serializable) where i = 4--rollback |

Здесь мы пытаемся удалить несуществующую строчку. В соответствии с тем, что наш запрос должен выдавать всегда один и тот же результат, менеджер блокировок должен обеспечить, что никакие другие транзакции не смогут вставить какое-либо значение в диапазон от 3 до 5, а также изменить граничные значения ключей на 4. Это достигается за счет установки разделяемой блокировки обновления диапазона.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| spid | dbid | ObjId | IndId | Type | Resource | Mode | Status |
| 55 | 8 | 2009058193 | 2 | KEY | (23005e3c905a)  | RangeS-U | GRANT |
| 55 | 8 | 2009058193 | 2 | PAG | 1:34 | IU | GRANT |

Если бы удаляемое значение было равно, например, 40, догадайтесь, какой диапазон был бы заблокирован менеджером блокировок. :) Он бы заблокировал весь диапазон от 9 до положительной бесконечности. А что если бы граничные значения ключа составляли бы не 3 и 5, а 9 и 500? Был бы заблокирован диапазон от 9 до 500. Таковы требования высшего уровня изоляции транзакций!

Чтобы еще лучше усвоить правила блокировки диапазонов, попробуйте выполнить такой запрос:

|  |
| --- |
| --set implicit\_transactions onselect \* from test\_key with (serializable) where i = 10--delete from test\_key with (serializable) where i = 4--rollback |

Он приведет к установлению разделяемой блокировки на диапазон от 9 до +INF, так что никакие операции вставки для этого диапазона не смогут быть выполнены.

Блокировки схемы данных

Последний тип блокировок, рассматриваемых в этой статье, связан с командами DDL и схемой данных (Schema). Блокировка схемы (Schema lock) предназначена для блокирования метаданных объектов базы данных. Типичной командой изменения таких данных может служить команда ALTER. Эта команда приводит к изменению системных таблиц метаданных, таких, как syscolumns, sysobjects, sysforeignkeys и других. Чтобы запретить другим транзакциям обращаться к модифицируемым ресурсам и их метаданным, введены блокировки схемы. Блокировки схемы бывают всего двух типов:

Блокировка стабильности схемы (Schema Stability Lock), обозначается Sch-S. Данный тип блокировки предназначен для гарантии неизменности метаданных, но не самих данных. Т.е. блокировка стабильности схемы – единственная из всех типов блокировок, совместимых с монопольной блокировкой. В основном она устанавливается при компиляции тела запроса или хранимой процедуры, на это время запрещается вносить изменения в схему данных, однако разрешается устанавливать любой тип блокировок на сами данные, с которыми будет работать компилируемый запрос.

Блокировка изменения схемы (Schema Modification Lock), обозначается Sch-M. Данный тип блокировки не совместим ни с каким другим типом, ни с самим собой. Устанавливается после внесения изменений в схему данных и снимается после завершения транзакции.

Рассмотрим пример. В первой сессии выполните следующий код:

|  |
| --- |
| begin tran alter table test add another\_field int |

В таблице 11 приведено содержимое таблицы syslockinfo для данной сессии.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| spid | dbid | ObjId | IndId | Type | Resource | Mode | Status |
| 55 | 8 | 0 | 0 | DB |  | S | GRANT |
| 55 | 8 | 1 | 0 | TAB |  | IX | GRANT |
| 55 | 8 | 3 | 0 | TAB |  | IX | GRANT |
| 55 | 8 | 3 | 2 | KEY | (31038b3afd68)  | X | GRANT |
| 55 | 8 | 1993058136 | 0 | TAB |  | Sch-M | GRANT |
| 55 | 8 | 1 | 1 | KEY | (5800c7965d9d)  | X | GRANT |
| 55 | 8 | 3 | 1 | KEY | (5b00f3711a81)  | X | GRANT |
| 55 | 8 | 0 | 0 | DB |  | S | GRANT |

Давайте проанализируем четыре выделенные строчки:

Наложена монопольная блокировка ключа на объект с идентификатором 3. Это не что иное, как таблица syscolumns. Более подробную информацию об этом ключе можно получить, выполнив такой запрос:

|  |
| --- |
| select \* from sysindexes where id = 3 and indid = 2 |

Наложена блокировка изменения схемы на объект с идентификатором 1993058136. Это не что иное, как таблица test.

Наложена монопольная блокировка ключа на объект с идентификатором 1. Это не что иное, как таблица sysobjects. Более подробную информацию об этом ключе можно получить, выполнив такой запрос:

|  |
| --- |
| select \* from sysindexes where id = 1 and indid = 1 |

Наложена монопольная блокировка ключа c идентификатором 1 на объект с идентификатором 3. Подробную информацию об этом ключе можно получить, выполнив такой запрос:

|  |
| --- |
| select \* from sysindexes where id = 3 and indid = 1 |

Еще один тип блокировки – Bulk Update – используется совместно с операциями массивного обновления. Он совместим только с блокировкой стабильности схемы и с самим собой.

Блокировки могут преобразовываться друг в друга по следующей схеме (рисунок 1).

Рисунок 1.

Таблица 12 показывает совместимость описанных типов блокировок друг с другом.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Тип | IS | S | U | IX | SIX | X | Sch-S | Sch-M | BU |
| IS | + | + | + | + | + | - | + | - | - |
| S | + | + | + | - | + | - | + | - | - |
| U | + | + | - | - | - | - | + | - | - |
| IX | + | + | - | + | - | - | + | - | - |
| SIX | + | + | - | - | - | - | + | - | - |
| X | - | - | - | - | - | - | + | - | - |
| Sch-S | + | + | + | + | + | + | + | - | + |
| Sch-M | - | - | - | - | - | - | - | - | - |
| BU | - | - | - | - | - | - | + | - | + |

Хинты

Как уже говорилось ранее, для каждой сессии может быть выбран используемый по умолчанию уровень изоляции, действующий при выполнении любых команд T-SQL. Ниже я покажу, как это сделать.

Рассмотрим синтаксис команды установки уровня изоляции по умолчанию для текущего соединения.

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL  { READ COMMITTED  | READ UNCOMMITTED  | REPEATABLE READ  | SERIALIZABLE  } |

READ UNCOMMITTED – устанавливает уровень изоляции транзакций, на котором решается проблема «грязной» записи;

READ COMMITTED – устанавливает уровень изоляции, на котором решается проблема «грязного» чтения;

REPEATABLE READ – устанавливает уровень изоляции, на котором решается проблема неповторяемого чтения;

SERIALIZABLE – устанавливает уровень изоляции, на котором решается проблема чтения фантомов.

Установка какого-либо значения остается действительной до конца сессии, или до тех пор, пока пользователь явно не изменит уровень изоляции транзакций.

Существует документированный способ просмотра текущего уровня изоляции с помощью команды DBCC USEROPTIONS, но лично я всегда пользуюсь более мощным, хотя и не документированным, средством. Чтобы узнать текущее значение изоляции, нужно воспользоваться таким скриптом:

|  |
| --- |
| dbcc traceon (3604)dbcc pss(1,53)dbcc traceoff (3604) |

Команда dbcc pss предназначена для вывода текущих настроек указанного в параметре соединения. Вот ее синтаксис:

|  |
| --- |
| DBCC pss( suid, spid, printopt = { 1 | 0 } ) |

suid – идентификатор пользователя (см. хранимую процедуру sp\_helpuser);

spid – идентификатор серверного процесса;

printopt – флаг вывода (0 – стандартная информация, 1 – расширенная). По умолчанию 0.

Эта команда выводит много интересной информации, но для нас главное – выяснить текущий уровень изоляции. Для этого нужно среди выведенных многочисленных переменных найти одну с названием isolation\_level. В моем случае она оказалась равна 0. Это говорит о том, что для данного соединения используется уровень изоляции по умолчанию, который, как я уже говорил, равен READ COMMITTED. Вот все значения, которые может принимать эта переменная:

0 – уровень изоляции по умолчанию (READ COMMITTED);

1 – уровень изоляции READ UNCOMMITTED;

2 – уровень изоляции READ COMMITTED;

3 – уровень изоляции REPEATABLE READ;

4 – уровень изоляции SERIALIZABLE.

Теперь рассмотрим, каким образом управлять уровнем изоляции транзакций на уровне отдельных команд. Вот синтаксис раздела FROM, который относится к нашей теме (все сказанное ниже относится не только к запросам, но и к командам изменения данных DELETE и UPDATE):

|  |
| --- |
| FROM { < table\_source > }<table\_source> ::=  table\_name [ [ AS ] table\_alias ] [ WITH ( < table\_hint > [ ,...n ] ) ]< table\_hint > ::=  { INDEX ( index\_val [ ,...n ] )  | FASTFIRSTROW | HOLDLOCK | NOLOCK | PAGLOCK | READCOMMITTED | READPAST | READUNCOMMITTED | REPEATABLEREAD | ROWLOCK | SERIALIZABLE | TABLOCK | TABLOCKX | UPDLOCK | XLOCK } |

Уровни READUNCOMMITTED и READCOMMITTED соответствуют одноименным уровням изоляции транзакций, только пишутся слитно. Блокировки при использовании этих уровней снимаются сразу после выполнения команды. В случае хинтов REPEATABLEREAD и SERIALIZABLE блокировки остаются до конца транзакции. Далее приводится подробное описание всех остальных хинтов.

FASTFIRSTROW – хинт не относится к блокировкам и уровням изоляции. Позволяет быстро выбрать первые строчки результирующего набора.

HOLDLOCK – аналогичен хинту SERIALIZABLE, т.е. устанавливает разделяемую блокировку диапазона индекса (если индекс существует) или простую разделяемую блокировку на страницу или таблицу. Оставлен для обратной совместимости.

NOLOCK – разрешается использовать только с командой SELECT. Аналогичен хинту READUNCOMMITTED, т.е. не накладывает никаких блокировок и игнорирует блокировки других транзакций.

PAGLOCK – пожелание менеджеру блокировок устанавливать блокировки на уровне страниц. Это пожелание выполняется очень редко.

READPAST – разрешается использовать только с командой SELECT. Позволяет выбрать только те строки, на которые не установлена монопольная блокировка. Другими словами, позволяет выбрать все не измененные строки.

ROWLOCK – пожелание менеджеру блокировок устанавливать блокировки на уровне строк. Это пожелание выполняется очень редко.

TABLOCK – позволяет установить на всю таблицу блокировку, тип которой зависит от команды, в которой этот хинт используется. Для команды SELECT будет установлена разделяемая блокировка на всю таблицу до тех пор, пока команда не выполнится, но не до конца транзакции. Действие хинта можно продлить с помощью HOLDLOCK. Например:

|  |
| --- |
| select \* from test with(tablock,holdlock) |

Установит разделяемую блокировку до конца транзакции.

TABLOCKX – устанавливает монопольную блокировку на всю таблицу до конца транзакции даже для команды SELECT.

UPDLOCK – разрешается использовать только с командой SELECT. Позволяет установить блокировку обновления вместо разделяемой блокировки. Действие блокировки продолжается до завершения транзакции.

XLOCK – Действие аналогично хинту UPDLOCK с той лишь разницей, что устанавливается монопольная блокировка на ресурс.

Теперь давайте рассмотрим примеры их использования. Итак, у нас имеется таблица test следующего содержания (таблица 13).

|  |  |
| --- | --- |
| i | n |
| 1 | alex |
| 2 | rosa |
| 3 | dima |

Начнем с монопольных блокировок. В первой сессии выполните команду:

|  |
| --- |
| update test set n = 'other' where i = 2 |

К тому же результату (наложение монопольной блокировки на строку) привела бы команда

|  |
| --- |
| select \* from test with(xlock) where i = 2 |

Теперь, во второй сессии:

|  |
| --- |
| select \* from test with (nolock) |

Вторая сессия не заблокировалась и выдала три строчки, включая измененную. Если бы мы не указали хинта nolock, сессия заблокировалась бы, так как по умолчанию установлен режим изоляции READ COMMITTED. Этому режиму необходима разделяемая блокировка, но так как монопольная блокировка не совместима ни с какой другой, менеджер блокировок заблокировал бы сессию.

Еще один интересный результат можно получить с помощью следующей команды (во второй сессии):

|  |
| --- |
| select \* from test with (readpast) |

Результаты ее выполнения зависят от того, какую команду мы выполнили в первой сессии. Если выполнялся запрос с хинтом xlock, результирующий набор будет содержать все строки таблицы. Однако если была выполнена команда UPDATE, SQL Server выдаст нам всего две строчки: 1 и 3. Такое поведение, признаться, для меня явилось неожиданностью, так как блокировки, накладываемые этими командами, полностью совпадают. У меня возникло предположение, что сервер каким-то образом помечает действительно измененные строки или вообще не вносит их в файл базы данных. Чтобы получить больше информации по этому поводу, воспользуемся очередной недокументированной возможностью, позволяющей просматривать физическое содержимое данных.

Но сначала я бы хотел рассмотреть содержимое колонки Resources, которую возвращает хранимая процедура sp\_lock. Значение это состоит из трех цифр, разделенных двоеточием, первое из которых означает номер файла в базе данных, второе – номер страницы и третье – номер строки. Таким образом, запись 1:31:2 означает вторую строку на странице 31 в первом файле базы данных.

Вот синтаксис команды, с помощью которой можно получить физическое содержимое таблицы:

|  |
| --- |
| DBCC PAGE ({dbid|dbname}, filenum, pagenum [,print option] [,cache]) |

dbid – идентификатор базы данных; dbname – название базы данных;

filenum – идентификатор файла базы данных;

pagenum – идентификатор страницы

print option – флаги вывода информации. Может принимать значения, приведенные в таблице 14.

|  |  |
| --- | --- |
| Значение | Описание |
| 0 | Установлен по умолчанию. Выводится только заголовок страницы. |
| 1 | Выводится заголовок страницы и содержимое каждой строчки. |
| 2 | Выводится заголовок страницы и полный дамп памяти. |
| 3 | Относительно новый флаг, который аналогичен флагу 1, с той разницей, что значения форматируются в удобочитаемый вид для каждой колонки. |

cache – флаг, указывающий, выбирать данные из кэша или нет. Если 1 (по умолчанию), данные выбираются из кэша, 0 – с диска.

Откроем новую сессию и выполним следующий набор команд (у меня база, в которой содержится таблица test, имеет идентификатор 8):

|  |
| --- |
| dbcc traceon (3604)dbcc page(8,1,31,3,1)dbcc traceoff (3604) |

Приведу результат, выводимый данной командой:

|  |
| --- |
| PAGE: (1:31)------------BUFFER:-------BUF @0x00D88500---------------bpage = 0x19548000 bhash = 0x00000000 bpageno = (1:31)bdbid = 8 breferences = 1 bstat = 0xbbspin = 0 bnext = 0x00000000 PAGE HEADER:------------Page @0x19548000----------------m\_pageId = (1:31) m\_headerVersion = 1 m\_type = 1m\_typeFlagBits = 0x0 m\_level = 0 m\_flagBits = 0x0m\_objId = 1993058136 m\_indexId = 0 m\_prevPage = (0:0)m\_nextPage = (0:0) pminlen = 8 m\_slotCnt = 3m\_freeCnt = 8026 m\_freeData = 247 m\_reservedCnt = 0m\_lsn = (6:181:2) m\_xactReserved = 0 m\_xdesId = (0:0)m\_ghostRecCnt = 0 m\_tornBits = 0 Allocation Status-----------------GAM (1:2) = ALLOCATED SGAM (1:3) = ALLOCATED PFS (1:1) = 0x61 MIXED\_EXT ALLOCATED 50\_PCT\_FULL DIFF (1:6) = CHANGEDML (1:7) = NOT MIN\_LOGGED Slot 0 Offset 0x60------------------Record Type = PRIMARY\_RECORD Record Attributes = NULL\_BITMAP VARIABLE\_COLUMNS 19548060: 00080030 00000001 02000002 15001100 0...............19548070: 656c6100 78 .alexi = 1 n = alex Slot 1 Offset 0xe1------------------Record Type = PRIMARY\_RECORD Record Attributes = NULL\_BITMAP VARIABLE\_COLUMNS 195480E1: 00080030 00000002 02000002 16001100 0...............195480F1: 68746f00 7265 .otheri = 2 n = other Slot 2 Offset 0x8a------------------Record Type = PRIMARY\_RECORD Record Attributes = NULL\_BITMAP VARIABLE\_COLUMNS 1954808A: 00080030 00000003 02000002 15001100 0...............1954809A: 6d696400 61 .dimai = 3 n = dima  |

Отсюда можно сделать вывод, что данные обновляются непосредственно, а менеджер блокировок следует какому-то специальному алгоритму, проверяющему лог транзакций (transaction log), при использовании хинта readpast.

На будущее хочу заметить, что тип страницы (выделен) может принимать несколько значений, из которых для нас важны два: 1 – для страниц данных, 2 – для страниц индексов.

Давайте продвигаться дальше. Так как в данный момент у нас заблокирована одна строка (вторая), попытаемся во второй сессии выполнить такой запрос:

|  |
| --- |
| --select \* from test with (readpast)select \* from test where i = 1 |

Ура! Наконец-то она заблокировалась! :) Это произошло потому, что в отсутствии индекса на таблице оптимизатор выполняет полное сканирование таблицы для поиска строк, удовлетворяющих заданному логическому условию. При этом он неизбежно натыкается на заблокированную строчку, что приводит к блокировке транзакции. Вот какие блокировки установлены на текущий момент (сокращенно, таблица 15):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ObjId | IndId | Type | Resource | Mode | Status |
| 1993058136 | 0 | PAG | 1:31 | IS | GRANT |
| 1993058136 | 0 | PAG | 1:31 | IX | GRANT |
| 1993058136 | 0 | TAB |  | IS | GRANT |
| 1993058136 | 0 | TAB |  | IX | GRANT |
| 1993058136 | 0 | RID | 1:31:01 | X | GRANT |
| 1993058136 | 0 | RID | 1:31:01 | S | WAIT |
| 1993058136 | 0 | PAG | 1:31 | IS | GRANT |
| 1993058136 | 0 | PAG | 1:31 | IX | GRANT |
| 1993058136 | 0 | TAB |  | IS | GRANT |
| 1993058136 | 0 | TAB |  | IX | GRANT |

Мы видим, что на первую строку наложена монопольная блокировка и какой-то процесс (а точнее, наша вторая сессия) хочет получить на эту же строку разделяемую блокировку. Чтобы решить эту проблему, нужен индекс. Но прежде нужно разобраться с текущей ситуацией.

Предположим, вы не имеете доступа к сессиям 54 и 55, которые в данный момент конкурируют в получении доступа к ресурсам, и не можете выполнить команды фиксации транзакции или отката. Единственным возможным выходом из сложившейся ситуации является убиение одного из процессов. Для получения подробной информации о процессе можно воспользоваться хранимой процедурой sp\_who и ее недокументированным аналогом sp\_who2. Последняя возвращает значительно больше информации, однако результаты выполнения обеих содержат колонку cmd, в которой хранится тип выполняющейся команды, если процесс заблокирован, или AWAITING COMMAND – если все нормально.

Для получения более подробной информации о выполняющейся команде можно использовать следующую команду:

|  |
| --- |
| DBCC INPUTBUFFER (55) |

Эта команда возвращает такой результат:

|  |
| --- |
| EventType Parameters EventInfo-------------- ---------- -----------------------------------------------Language Event 0 --SET IMPLICIT\_TRANSACTIONS on--select \* from test where i = 1 |

Эта команда может использоваться для просмотра первых 255 символов команд для заданной сессии.

После того, как выяснится жертва, можно вызывать команду KILL.

Однако все это не про нас, вернемся к первой сессии (с номером 54) и отменим транзакцию:

|  |
| --- |
| --SET IMPLICIT\_TRANSACTIONS on --update test set n = 'other' where i = 2rollback |

Вторая сессия моментально разблокируется, после чего в ней также нужно откатить транзакцию:

|  |
| --- |
| --SET IMPLICIT\_TRANSACTIONS on --select \* from test with (readpast)--select \* from test where i = 1rollback |

Давайте создадим некластерный индекс и выполним те же самые действия. Откройте новую сессию и выполните следующую команду:

|  |
| --- |
| create nonclustered index \_i\_idx on test(i) |

Теперь в первой сессии выполним обновление записи:

|  |
| --- |
| --SET IMPLICIT\_TRANSACTIONS on update test set n = 'other' where i = 2--rollback |

А во второй – запрос:

|  |
| --- |
| --SET IMPLICIT\_TRANSACTIONS on --select \* from test with (readpast)select \* from test with (xlock) where i = 1--rollback |

Чтобы увидеть блокировки второй сессии, я поставил хинт xlock. Запрос не заблокируется, давайте посмотрим содержимое syslockinfo (таблица 16).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| spid | dbid | ObjId | IndId | Type | Resource | Mode | Status |
| 55 | 8 | 1993058136 | 0 | RID | 1:31:00 | X | GRANT |
| 55 | 8 | 1993058136 | 0 | PAG | 1:31 | IX | GRANT |
| 54 | 8 | 1993058136 | 0 | PAG | 1:31 | IX | GRANT |
| 55 | 8 | 1993058136 | 3 | PAG | 1:29 | IX | GRANT |
| 55 | 8 | 1993058136 | 0 | TAB |  | IX | GRANT |
| 54 | 8 | 1993058136 | 0 | TAB |  | IX | GRANT |
| 54 | 8 | 1993058136 | 0 | RID | 1:31:01 | X | GRANT |
| 55 | 8 | 1993058136 | 3 | KEY | (21001d31a802)  | X | GRANT |

Как видим, первая сессия наложила монопольную блокировку на первую строку 1:31:01, а вторая – на нулевую 1:31:00. Кроме этого, вторая сессия наложила монопольную блокировку на ключ (последняя строка) и монопольную блокировку намерений на страницу, где располагается индекс (1:29).

Чтобы действительно убедиться, что индекс создан именно на этой странице, нужно выполнить очередную недокументированную команду:

|  |
| --- |
| extentinfo [({'database\_name'| dbid} [, {'table\_name' | table\_id} [, {'index\_name' | index\_id}]])] |

dbid – идентификатор базы данных; dbname – название базы данных;

table\_id – идентификатор таблицы; table\_name – название таблицы;

index\_id – идентификатор индекса; index\_name – название индекса.

|  |
| --- |
| ПРИМЕЧАНИЕДля получения полного списка команд DBCC (Database Console Commands) можно воспользоваться такой командой:dbcc traceon (2520)dbcc help('?')dbcc traceoff (2520)Чтобы получить синтаксис конкретной команды, например, page, выполните такую последовательность:dbcc traceon (2520)dbcc help('page')dbcc traceoff (2520) |

В нашем случае следующий набор команд покажет все страницы таблицы test:

|  |
| --- |
| dbcc traceon (3604)dbcc extentinfo(8,'test')dbcc traceoff (3604) |

В колонке page\_id указаны идентификаторы страниц, а в колонке index\_id – идентификаторы индекса. Просмотрим содержимое страницы индекса:

|  |
| --- |
| dbcc traceon (3604)dbcc page(8,1,29,1,1)dbcc traceoff (3604) |

Вот результат работы данной команды:

|  |
| --- |
| PAGE: (1:29)------------BUFFER:-------BUF @0x00DA2740---------------bpage = 0x1A25A000 bhash = 0x00000000 bpageno = (1:29)bdbid = 8 breferences = 1 bstat = 0x9bspin = 0 bnext = 0x00000000 PAGE HEADER:------------Page @0x1A25A000----------------m\_pageId = (1:29) m\_headerVersion = 1 m\_type = 2m\_typeFlagBits = 0x0 m\_level = 0 m\_flagBits = 0x4m\_objId = 1993058136 m\_indexId = 3 m\_prevPage = (0:0)m\_nextPage = (0:0) pminlen = 13 m\_slotCnt = 3m\_freeCnt = 8042 m\_freeData = 144 m\_reservedCnt = 0m\_lsn = (6:181:20) m\_xactReserved = 0 m\_xdesId = (0:0)m\_ghostRecCnt = 0 m\_tornBits = 839385609 Allocation Status-----------------GAM (1:2) = ALLOCATED SGAM (1:3) = ALLOCATED PFS (1:1) = 0x60 MIXED\_EXT ALLOCATED 0\_PCT\_FULL DIFF (1:6) = CHANGEDML (1:7) = NOT MIN\_LOGGED DATA:-----Slot 0, Offset 0x60-------------------Record Type = INDEX\_RECORD Record Attributes = NULL\_BITMAP 1A25A060: 00000116 00001f00 00000100 00000300 ................Slot 1, Offset 0x70-------------------Record Type = INDEX\_RECORD Record Attributes = NULL\_BITMAP 1A25A070: 00000216 00001f00 01000100 00000300 ................Slot 2, Offset 0x80-------------------Record Type = INDEX\_RECORD Record Attributes = NULL\_BITMAP 1A25A080: 00000316 00001f00 02000100 00000300 ................ |

Так как тип страницы, указанный в заголовке, равен 2, данная страница является индексной. Кроме этого, значение m\_indexId равно идентификатору индекса.

|  |
| --- |
| ПРИМЕЧАНИЕИнформацию обо всех индексах данной таблицы можно получить с помощью такого запроса: select \* from sysindexes where id = object\_id('test') |

На этом я закончу рассматривать хинты SQL Server’а и блокировки и перейду к проблеме взаимоблокировок (deadlocks). Не забудьте завершать транзакции в 54 и 55 сессиях.

|  |
| --- |
| СОВЕТДля определения текущего количества незавершенных транзакций можно воспользоваться функцией @@trancount. |

Взаимоблокировки

В процессе работы параллельных транзакций, обращающихся к одним и тем же ресурсам, возможно возникновение проблемы взаимоблокировок (deadlock), которые также называют тупиковыми блокировками. Если транзакции обращаются к ресурсам последовательно, вероятность возникновения взаимоблокировки очень мала, однако если порядок обращения транзакций к общим ресурсам разный, возможность возникновения мертвой блокировки повышается многократно.

Рассмотрим суть взаимоблокировки на двух транзакциях (t1 и t2), обращающихся к двум ресурсам (A и B). Предположим, что транзакция t1 блокирует ресурс A, после чего транзакция t2 блокирует ресурс B. Если теперь транзакция t1 попытается получить доступ к ресурсу B, а транзакция t2 – к ресурсу А, возникнет взаимоблокировка. В таком состоянии транзакции не могут разрешить конфликт и переходят в состояние ожидания на неопределенно долгий срок. Чтобы изменить время ожидания ресурса для текущего соединения, используется следующая команда:

|  |
| --- |
| SET LOCK\_TIMEOUT timeout\_period |

timeout\_period задается в миллисекундах. Если указывается значение –1, менеджер транзакций будет ждать освобождения ресурса неопределенно долгое время.

В SQL Server’е предусмотрен механизм автоматического определения взаимоблокировок и разрешения конфликтов. Для этого предназначен специальный серверный процесс LOCK MONITOR, который просыпается каждые 5 секунд [2]. Он проверяет наличие двух и более ожидающих транзакции и вычисляет зависимости между ними. Если оказывается, что транзакции зависят друг от друга, т.е. для получения блокировки одной из них необходимо снять блокировку другой транзакции, фиксируется факт возникновения взаимоблокировки. Единственный способ устранения проблемы заключается в отмене одной из транзакций. Но какой? Для каждой транзакции вводится понятие цены или приоритета. Это значение задается на уровне соединения следующей командой:

|  |
| --- |
| SET DEADLOCK\_PRIORITY { LOW | NORMAL | @deadlock\_var } |

Задание меньшего приоритета для соединения приводит к откату транзакций при возникновении взаимоблокировок с кодом ошибки 1205.

Давайте рассмотрим простой пример взаимоблокировки. Для большей наглядности я продемонстрирую его на одной таблице из трех (!) строк.

Итак, мы продолжаем работать с нашей таблицей test, на поле i которой создан некластерный индекс.

В первой сессии выполним такой запрос:

|  |
| --- |
| SET IMPLICIT\_TRANSACTIONS onselect \* from test with(updlock) where i = 2 |

Здесь мы наложили блокировку обновления на вторую строку таблицы. Теперь во второй сессии введем следующий код:

|  |
| --- |
| SET IMPLICIT\_TRANSACTIONS onselect \* from test with (updlock) where i = 1 |

Та же блокировка обновления накладывается на первую строчку. Перейдем снова к первой сессии:

|  |
| --- |
| --SET IMPLICIT\_TRANSACTIONS on--select \* from test with(updlock) where i = 2update test set n = (select n from test where i = 2) where i = 1 |

Так как мы пытаемся получить на первую строку монопольную блокировку, которая несовместима с текущей блокировкой обновления, сессия блокируется.

Теперь во второй сессии введем похожую команду:

|  |
| --- |
| --SET IMPLICIT\_TRANSACTIONS on--select \* from test with (updlock) where i = 1update test set n = (select n from test where i = 1) where i = 2 |

Вот она, взаимоблокировка! Обе сессии заблокированы. Однако наше ликование будет недолгим: примерно через 5 секунд менеджер блокировок сообразит, что возникла взаимоблокировка и откатит первую транзакцию.

|  |
| --- |
| Server: Msg 1205, Level 13, State 50, Line 3Transaction (Process ID 58) was deadlocked on lock resources with another process and has been chosenas the deadlock victim. Rerun the transaction. |

Конечно, пример несколько надуманный, зато он хорошо иллюстрирует проблему: на пустом месте можно получить взаимоблокировку даже на табличке из двух строк.

При написании хранимых процедур, которые и являются основным источником взаимных блокировок, необходимо руководствоваться следующим простым правилом: если две и более транзакций используют совместные ресурсы, порядок наложения однотипных блокировок на эти ресурсы должен быть одинаковым.

Это правило помогает, но не во всех случаях. Рассмотрим такой пример. В первой сессии выполняем команду:

|  |
| --- |
| set implicit\_transactions on select \* from test with (repeatableread) |

Это приводит к наложению разделяемых блокировок на все строки таблицы test. Во второй сессии выполняем тот же запрос:

|  |
| --- |
| set implicit\_transactions on select \* from test with (repeatableread) |

Так как разделяемая блокировка совместима сама с собой, запрос выполняется нормально.

Блокировки для первой строки приведены в таблице 17.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| spid | dbid | ObjId | IndId | Type | Resource | Mode | Status |
| 55 | 8 | 1993058136 | 0 | RID | 1:31:00 | S | GRANT |
| 57 | 8 | 1993058136 | 0 | RID | 1:31:00 | S | GRANT |

Нас интересует именно первая строка, так как сейчас мы попытаемся в первой сессии изменить данные:

|  |
| --- |
| --set implicit\_transactions on --select \* from test with (repeatableread)update test set n = 'other' where i = 1 |

Сессия блокируется, так как монопольная блокировка, необходимая для выполнения изменения, не совместима с разделяемой. Теперь блокировки выглядят так, как показано в таблице 18.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| spid | dbid | ObjId | IndId | Type | Resource | Mode | Status |
| 55 | 8 | 1993058136 | 0 | RID | 1:31:00 | S | GRANT |
| 57 | 8 | 1993058136 | 0 | RID | 1:31:00 | U | GRANT |
| 57 | 8 | 1993058136 | 0 | RID | 1:31:00 | X | CNVT |

Обнаружив нужную строчку, сервер пытается преобразовать текущую разделяемую блокировку в блокировку обновления, что ему удается. Это делается для того, чтобы после нахождения нужного ресурса в процессе подготовительных операций перед обновлением никакая другая транзакция не смогла получить блокировку на изменение (блокировку обновления или монопольную блокировку). Затем, непосредственно перед обновлением, сервер пытается преобразовать текущую блокировку обновления в монопольную, что ему, естественно, не удается.

Если теперь выполнить во второй сессии ту же команду:

|  |
| --- |
| --set implicit\_transactions on --select \* from test with (repeatableread)update test set n = 'other' where i = 1 |

мы получим мертвую блокировку. Причем во второй сессии транзакция даже не сможет преобразовать разделяемую блокировку в блокировку обновления, так как блокировка обновления не совместима сама с собой.

Чтобы избавиться от подобных ситуаций, нужно устанавливать при запросе данных вместо разделяемой блокировки сразу блокировку обновления. В этом случае вторая сессия будет заблокирована на команде выборки данных до завершения транзакции в первой сессии. Однако и при установке блокировки обновления нельзя быть уверенным, что не произойдет взаимного блокирования транзакций. Если во второй сессии просто выполнить запрос с уровнем изоляции REPEATABLE READ, на все строки опять будет установлена разделяемая блокировка, и при попытке обновления записи в первой сессии, транзакция будет заблокирована. Единственным выходом из ситуации, когда происходит выполнение транзакций с разными уровнями изоляции, будет наложение монопольных блокировок на все строки при запросе данных в первой сессии. Хотя такой метод строго не рекомендуется, он возможен, благодаря наличию хинта xlock. Если вносимые транзакцией изменения будут затрагивать более 70% таблицы, есть смысл вместо xlock установить хинт tablockx, хотя менеджер блокировок достаточно умен, чтобы самостоятельно выполнить эскалацию блокировок.

**Заключение**

В заключение я хочу рассмотреть вопросы, которые не были затронуты в данной статье, но могут иметь определенный интерес.

Владельцы блокировки

Возможно, вы не задумывались над этим вопросом, или думали, что владельцем всех блокировок является менеджер блокировок, однако это не так. У каждой блокировки есть владелец, и его можно найти в таблице syslockinfo в поле req\_ownertype. Это поле может принимать три значения: 1, 2 и 3. Если значение равно 1, владельцем блокировки является транзакция – это самая распространенная ситуация. Если req\_ownertype равен 2 – владельцем является курсор, созданный с опцией SCROLL\_LOCKS. И, наконец, значение 3 говорит о том, что владельцем транзакции является сессия. Как правило, объектом подобной блокировки является база данных, а сама блокировка накладывается в момент выполнения команды use [database] и снимается, когда выполняется другая команда use, или завершается сессия.

Опции индекса

По умолчанию SQL Server выбирает наиболее подходящий, по его мнению, уровень детализации блокировок и выполняет эскалацию блокировок при увеличении их количества. Вы можете задать другую стратегию блокировки непосредственно в самом запросе, с помощью хинтов, а можете (только для индексов) указать заранее степень детализации объектов блокировок. Делается это с помощью хранимой процедуры sp\_indexoption. Вот ее синтаксис:

|  |
| --- |
| sp\_indexoption [ @IndexNamePattern = ] 'index\_name'  , [ @OptionName = ] 'option\_name'  , [ @OptionValue = ] 'value' |

IndexNamePattern – Имя индекса.

OptionName – Имя опции. Может принимать значения, перечисленные в таблице 19.

|  |  |
| --- | --- |
| Имя опции | Описание |
| AllowRowLocks | Если установлена в TRUE, разрешает использования блокировок на уровне строк. |
| AllowPageLocks | Если установлена в TRUE, разрешает использования блокировок на уровне страниц. |
| DisAllowRowLocks | Если установлена в TRUE, запрещает использование блокировок на уровне строк. |
| DisAllowPageLocks | Если установлена в TRUE, запрещает использование блокировок на уровне страниц. |

Булево значение.

Рассмотрим пример ее использования. Следующий запрос установит около сотни блокировок, в чем можно убедиться, воспользовавшись хранимой процедурой sp\_lock.

|  |
| --- |
| set implicit\_transactions onselect City from Customers with (repeatableread) |

Однако если перед этим запросов вызвать хранимую процедуру sp\_indexoption следующим образом:

|  |
| --- |
| EXEC sp\_indexoption 'Customers.City', 'allowrowlocks', false |

Будет установлена всего одна (!) разделяемая блокировка на таблицу Customers.

Блокировки приложений

В SQL Server 2000 появились две новые процедуры, предназначенные для работы с пользовательскими блокировками. Пользовательские блокировки позволяют решить проблему доступа к тем ресурсам, на которые не распространяется действие менеджера блокировок. Например, когда вы хотите защитить отдельный столбец в таблице или вызов хранимой процедуры, пользовательские блокировки являются лучшим выходом.

|  |
| --- |
| ПРИМЕЧАНИЕВозможно также написание расширенных хранимых процедур, в которых использовались бы механизмы синхронизации операционной системы. До выхода SQL Server 2000 это был единственный метод. |

Рассмотрим синтаксис процедур.

|  |
| --- |
| sp\_getapplock [ @Resource = ] 'resource\_name',  [ @LockMode = ] 'lock\_mode'  [ , [ @LockOwner = ] 'lock\_owner' ]  [ , [ @LockTimeout = ] 'value' ] |

Resource – уникальное имя ресурса, аналогично уникальному имени объекта мьютекса или события.

LockMode – Тип запрашиваемой блокировки. Может принимать значения Shared, Update, Exclusive, IntentExclusive и IntentShared.

LockOwner – владелец блокировки. По умолчанию – transaction. Может принимать значения transaction или session. При указании владельца Session, блокировка снимается либо при завершении сессии, либо при завершении работы SQL Server’а, либо при вызове процедуры sp\_releaseapplock.

LockTimeout – значение в миллисекундах, в течение которого транзакция будет ожидать ресурса, если не сможет получить доступа сразу. По умолчанию берется значение, которое возвращает функция @@LOCK\_TIMEOUT.

Чтобы убедиться, что ресурс захвачен, необходимо проверить возвращаемое значение функции. Если оно больше или равно 0, ресурс захвачен, если нет – произошла ошибка. Все возможные значения приведены в таблице 20.

|  |  |
| --- | --- |
| Значение | Описание |
| 0 | Ресурс успешно захвачен. |
| 1 | Ресурс был захвачен после определенного ожидания. |
| -1 | Ресурс захвачен не был, произошел возврат по таймауту. |
| -2 | Запрос на ресурс был прерван. |
| -3 | Запрос на ресурс был прерван вследствие возникновения мертвой блокировки.  |
| -999 | В процедуру были переданы неверные параметры. |

После успешного захвата ресурса вы можете быть уверены, что обладаете тем типом доступа, который запросили. После работы с ресурсом, необходимо вызвать процедуру sp\_releaseapplock. Вот ее синтаксис:

|  |
| --- |
| sp\_releaseapplock [ @Resource = ] 'resource\_name'  [ , [ @LockOwner = ] 'lock\_owner' ] |

Resource – уникальное имя ресурса, аналогично уникальному имени объекта мьютекса или события.

LockOwner – владелец блокировки. По умолчанию – transaction. Может принимать значения transaction или session.

Эта процедура должна быть вызвана столько раз, сколько вызывалась процедура sp\_getapplock.

Примеры использования данной связки имеются в MSDN.

На этом я заканчиваю статью, надеюсь, она оказалась для вас интересной и полезной в практическом плане.

**Список литературы**

Microsoft SQL Server 2000, Е.В Мамаев, 2001 г.

Inside Microsoft SQL Server 2000, Kalen Delaney, 2001 г.