### ACID

**Атомарность** гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все её подоперации, либо не выполнено ни одной. Поскольку на практике невозможно одновременно и атомарно выполнить всю последовательность операций внутри транзакции, вводится понятие «отката» (rollback): если транзакцию не удаётся полностью завершить, результаты всех её до сих пор произведённых действий будут отменены и система вернётся во «внешне исходное» состояние — со стороны будет казаться, что транзакции и не было. (Естественно, счётчики, индексы и другие внутренние структуры могут измениться, но, если СУБД запрограммирована без ошибок, это не повлияет на внешнее её поведение.)

**Согласованость**. Транзакция, достигающая своего нормального завершения (EOT — end of transaction, завершение транзакции) и, тем самым, фиксирующая свои результаты, сохраняет согласованность базы данных. Другими словами, каждая успешная транзакция по определению фиксирует только допустимые результаты. Это условие является необходимым для поддержки четвёртого свойства.

Согласованность является более широким понятием. Например, в банковской системе может существовать требование равенства суммы, списываемой с одного счёта, сумме, зачисляемой на другой. Это бизнес-правило и оно не может быть гарантировано только проверками целостности, его должны соблюсти программисты при написании кода транзакций. Если какая-либо транзакция произведёт списание, но не произведёт зачисление, то система останется в некорректном состоянии и свойство согласованности будет нарушено.

Наконец, ещё одно замечание касается того, что в ходе выполнения транзакции согласованность не требуется. В нашем примере, списание и зачисление будут, скорее всего, двумя разными подоперациями и между их выполнением внутри транзакции будет видно несогласованное состояние системы. Однако не нужно забывать, что при выполнении требования изоляции, никаким другим транзакциям эта несогласованность не будет видна. А атомарность гарантирует, что транзакция либо будет полностью завершена, либо ни одна из операций транзакции не будет выполнена. Тем самым эта промежуточная несогласованность является скрытой.

**Изолированоость**. Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат. Изолированность — требование дорогое, поэтому в реальных БД существуют режимы, не полностью изолирующие транзакцию (уровни изолированности Repeatable Read и ниже).

**Надежность**. Независимо от проблем на нижних уровнях (к примеру, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершённой транзакцией, должны остаться сохранёнными после возвращения системы в работу. Другими словами, если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены из-за какого-либо сбоя.

### Проблемы параллельного доступа с использованием транзакций

При параллельном выполнении транзакций возможны следующие проблемы:

**потерянное обновление** (англ. lost update) — при одновременном изменении одного блока данных разными транзакциями одно из изменений теряется;

**«грязное» чтение** (англ. dirty read) — чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится);

**неповторяющееся чтение** (англ. non-repeatable read) — при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными;

**фантомное чтение** (англ. phantom reads) — одна транзакция в ходе своего выполнения несколько раз выбирает множество строк по одним и тем же критериям. Другая транзакция в интервалах между этими выборками добавляет или удаляет строки или изменяет столбцы некоторых строк, используемых в критериях выборки первой транзакции, и успешно заканчивается. В результате получится, что одни и те же выборки в первой транзакции дают разные множества строк.

Рассмотрим ситуации, в которых возможно возникновение данных проблем.

### Потерянное обновление

Ситуация, когда при одновременном изменении одного блока данных разными транзакциями одно из изменений теряется.

Предположим, имеются две транзакции, выполняемые одновременно:

|  |  |
| --- | --- |
| **Транзакция 1** | **Транзакция 2** |
| UPDATE tbl1 SET f2=f2+20 WHERE f1=1; | UPDATE tbl1 SET f2=f2+25 WHERE f1=1; |

В обеих транзакциях изменяется значение поля f2, по их завершении значение поля должно быть увеличено на 45. В действительности может возникнуть следующая последовательность действий:

Обе транзакции одновременно читают текущее состояние поля. Точная физическая одновременность здесь не обязательна, достаточно, чтобы вторая по порядку операция чтения выполнилась до того, как другая транзакция запишет свой результат.

Обе транзакции вычисляют новое значение поля, прибавляя, соответственно, 20 и 25 к ранее прочитанному значению.

Транзакции пытаются записать результат вычислений обратно в поле f2. Поскольку физически одновременно две записи выполнить невозможно, в реальности одна из операций записи будет выполнена раньше, другая позже. При этом вторая операция записи перезапишет результат первой.

В результате значение поля f2 по завершении обеих транзакций может увеличиться не на 45, а на 20 или 25, то есть одна из изменяющих данные транзакций «пропадёт».

### «Грязное» чтение

Чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится).

Предположим, имеются две транзакции, открытые различными приложениями, в которых выполнены следующие SQL-операторы:

|  |  |
| --- | --- |
| **Транзакция 1** | **Транзакция 2** |
|  | SELECT f2 FROM tbl1 WHERE f1=1; |
| UPDATE tbl1 SET f2=f2+1 WHERE f1=1; |  |
|  | SELECT f2 FROM tbl1 WHERE f1=1; |
| ROLLBACK WORK; |  |

В транзакции 1 изменяется значение поля f2, а затем в транзакции 2 выбирается значение этого поля. После этого происходит откат транзакции 1. В результате значение, полученное второй транзакцией, будет отличаться от значения, хранимого в базе данных.

**Неповторяющееся чтение**

Ситуация, когда при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными.

Предположим, имеются две транзакции, открытые различными приложениями, в которых выполнены следующие SQL-операторы:

|  |  |
| --- | --- |
| **Транзакция 1** | **Транзакция 2** |
|  | SELECT f2 FROM tbl1 WHERE f1=1; |
| UPDATE tbl1 SET f2=f2+1 WHERE f1=1; |  |
| COMMIT; |  |
|  | SELECT f2 FROM tbl1 WHERE f1=1; |

В транзакции 2 выбирается значение поля f2, затем в транзакции 1 изменяется значение поля f2. При повторной попытке выбора значения из поля f2 в транзакции 2 будет получен другой результат. Эта ситуация особенно неприемлема, когда данные считываются с целью их частичного изменения и обратной записи в базу данных.

### Фантомное чтение

Ситуация, когда при повторном чтении в рамках одной транзакции одна и та же выборка дает разные множества строк.

Предположим, имеется две транзакции, открытые различными приложениями, в которых выполнены следующие SQL-операторы:

|  |  |
| --- | --- |
| **Транзакция 1** | **Транзакция 2** |
|  | SELECT SUM(f2) FROM tbl1; |
| INSERT INTO tbl1 (f1,f2) VALUES (15,20); |  |
| COMMIT; |  |
|  | SELECT SUM(f2) FROM tbl1; |

В транзакции 2 выполняется SQL-оператор, использующий все значения поля f2. Затем в транзакции 1 выполняется вставка новой строки, приводящая к тому, что повторное выполнение SQL-оператора в транзакции 2 выдаст другой результат. Такая ситуация называется фантомным чтением. От неповторяющегося чтения оно отличается тем, что результат повторного обращения к данным изменился не из-за изменения/удаления самих этих данных, а из-за появления новых (фантомных) данных.

### Уровни изоляции

Под «**уровнем изоляции транзакций**» понимается степень обеспечиваемой внутренними механизмами СУБД (то есть не требующей специального программирования) защиты от всех или некоторых видов вышеперечисленных несогласованностей данных, возникающих при параллельном выполнении транзакций. Стандарт SQL-92 определяет шкалу из четырёх уровней изоляции: Read uncommitted, Read committed, Repeatable read, Serializable. Первый из них является самым слабым, последний — самым сильным, каждый последующий включает в себя все предыдущие.

**Read uncommitted** (чтение незафиксированных данных)

Низший (нулевой) уровень изоляции. Он гарантирует только отсутствие потерянных обновлений. Если несколько параллельных транзакций пытаются изменять одну и ту же строку таблицы, то в окончательном варианте строка будет иметь значение, определенное всем набором успешно выполненных транзакций. При этом возможно считывание не только логически несогласованных данных, но и данных, изменения которых ещё не зафиксированы.

Типичный способ реализации данного уровня изоляции — блокировка данных на время выполнения команды изменения, что гарантирует, что команды изменения одних и тех же строк, запущенные параллельно, фактически выполнятся последовательно, и ни одно из изменений не потеряется. Транзакции, выполняющие только чтение, при данном уровне изоляции никогда не блокируются (select во второй транзакции НЕ виснет после update в активной первой транзакции, потому во второй возможно чтение того, что не подтвердится в первой).

**Read committed** (чтение зафиксированных данных)

Большинство промышленных СУБД, в частности, [Microsoft SQL Server](https://ru.wikipedia.org/wiki/Microsoft_SQL_Server), [PostgreSQL](https://ru.wikipedia.org/wiki/PostgreSQL) и [Oracle](https://ru.wikipedia.org/wiki/Oracle_%28%D0%A1%D0%A3%D0%91%D0%94%29), по умолчанию используют именно этот уровень. На этом уровне обеспечивается защита от чернового, «грязного» чтения, тем не менее, в процессе работы одной транзакции другая может быть успешно завершена и сделанные ею изменения зафиксированы. В итоге первая транзакция будет работать с другим набором данных.

Реализация завершённого чтения может основываться на одном из двух подходов: блокировании или версионности.

Блокирование читаемых и изменяемых данных.

Заключается в том, что читающая транзакция блокирует читаемые данные в разделяемом (shared) режиме, в результате чего параллельная транзакция, пытающаяся изменить эти данные, приостанавливается, а пишущая транзакция блокирует изменяемые данные для читающих транзакций, работающих на уровне read committed или более высоком, до своего завершения, препятствуя, таким образом, «грязному» чтению (select во второй транзакции виснет пока не подтвердится update в первой транзакции).

Сохранение нескольких версий параллельно изменяемых строк.

При каждом изменении строки СУБД создаёт новую версию этой строки, с которой продолжает работать изменившая данные транзакция, в то время как любой другой «читающей» транзакции возвращается последняя зафиксированная версия. Преимущество такого подхода в том, что он обеспечивает бо́льшую скорость, так как предотвращает блокировки. Однако он требует, по сравнению с первым, существенно бо́льшего расхода оперативной памяти, которая тратится на хранение версий строк. Кроме того, при параллельном изменении данных несколькими транзакциями может создаться ситуация, когда несколько параллельных транзакций произведут несогласованные изменения одних и тех же данных (поскольку блокировки отсутствуют, ничто не помешает это сделать). Тогда та транзакция, которая зафиксируется первой, сохранит свои изменения в основной БД, а остальные параллельные транзакции окажется невозможно зафиксировать (так как это приведёт к потере обновления первой транзакции). Единственное, что может в такой ситуации СУБД — это откатить остальные транзакции и выдать сообщение об ошибке «Запись уже изменена».

Конкретный способ реализации выбирается разработчиками СУБД, а в ряде случае может настраиваться. Так, по умолчанию MS SQL использует блокировки, но (в версии 2005 и выше) при установке параметра READ\_COMMITTED\_SNAPSHOT базы данных переходит на стратегию версионности, Oracle исходно работает только по версионной схеме.

**Repeatable read** (повторяемость чтения)

Уровень, при котором читающая транзакция «не видит» изменения данных, которые были ею ранее прочитаны. При этом никакая другая транзакция не может изменять данные, читаемые текущей транзакцией, пока та не окончена.

Блокировки в разделяющем режиме применяются ко всем данным, считываемым любой инструкцией транзакции, и сохраняются до её завершения. Это запрещает другим транзакциям изменять строки, которые были считаны незавершённой транзакцией. Однако другие транзакции могут вставлять новые строки, соответствующие условиям поиска инструкций, содержащихся в текущей транзакции. При повторном запуске инструкции текущей транзакцией будут извлечены новые строки, что приведёт к фантомному чтению. Учитывая то, что разделяющие блокировки сохраняются до завершения транзакции, а не снимаются в конце каждой инструкции, степень параллелизма ниже, чем при уровне изоляции READ COMMITTED. Поэтому пользоваться данным и более высокими уровнями транзакций без необходимости обычно не рекомендуется.

**Serializable** (упорядочиваемость)

Самый высокий уровень изолированности; транзакции полностью изолируются друг от друга, каждая выполняется так, как будто параллельных транзакций не существует. Только на этом уровне параллельные транзакции не подвержены эффекту «фантомного чтения».

### Поддержка изоляции транзакций в реальных СУБД

СУБД, обеспечивающие транзакционность, не всегда поддерживают все четыре уровня, а также могут вводить дополнительные. Возможны также различные нюансы в обеспечении изоляции.

Так, Oracle в принципе не поддерживает нулевой уровень, так как его реализация транзакций исключает «грязные чтения», и формально не позволяет устанавливать уровень Repeatable read, то есть поддерживает только Read committed (по умолчанию) и Serializable. При этом на уровне отдельных команд он, фактически, гарантирует повторяемость чтения (если команда SELECT в первой транзакции выбирает из базы набор строк, и в это время параллельная вторая транзакция изменяет какие-то из этих строк, то результирующий набор, полученный первой транзакцией, будет содержать неизменённые строки, как будто второй транзакции не было). Также Oracle поддерживает так называемые READ-ONLY транзакции, которые соответствуют Serializable, но при этом не могут сами изменять данные.

Microsoft SQL Server поддерживает все четыре стандартных уровня изоляции транзакций, а дополнительно — уровень SNAPSHOT, находящийся между Repeatable read и Serialized. Транзакция, работающая на данном уровне, видит только те изменения данных, которые были зафиксированы до её запуска, а также изменения, внесённые ею самой, то есть ведёт себя так, как будто получила при запуске моментальный снимок данных БД и работает с ним.

### Поведение при различных уровнях изолированности

«+» — предотвращает, «-» — не предотвращает.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Уровень изоляции** | **Фантомное чтение** | **Неповторяющееся чтение** | **«Грязное» чтение** | **Потерянное обновление** |
| SERIALIZABLE | + | + | + | + |
| REPEATABLE READ | - | + | + | + |
| READ COMMITTED | - | - | + | + |
| READ UNCOMMITTED | - | - | - | + |

**Блокировки**

Повышение эффективности работы при использовании небольших транзакций связано с тем, что при выполнении транзакции сервер накладывает на данные блокировки.

Если для выполнения некоторой транзакции необходимо, чтобы некоторый объект базы данных не изменялся без ведома этой транзакции, такой объект блокируется.

Блокировкой называется временное ограничение на выполнение некоторых операций обработки данных. Блокировка может быть наложена как на отдельную строку таблицы, так и на всю базу данных. Управлением блокировками на сервере занимается менеджер блокировок, контролирующий их применение и разрешение конфликтов. Транзакции и блокировки тесно связаны друг с другом. Транзакции накладывают блокировки на данные, чтобы обеспечить выполнение требований ACID. Без использования блокировок несколько транзакций могли бы изменять одни и те же данные.

Блокировка представляет собой метод управления параллельными процессами, при котором объект БД не может быть модифицирован без ведома транзакции, т.е. происходит блокирование доступа к объекту со стороны других транзакций, чем исключается непредсказуемое изменение объекта. Различают два вида блокировки:

* блокировка записи – транзакция блокирует строки в таблицах таким образом, что запрос другой транзакции к этим строкам будет отменен;
* блокировка чтения – транзакция блокирует строки так, что запрос со стороны другой транзакции на блокировку записи этих строк будет отвергнут, а на блокировку чтения – принят.

То есть другими словами если клиент хочет читать данные, то другие клиенты тоже могут читать данные, но никто не может записывать, пока первый клиент  не закончит чтение (read lock), и если клиент  хочет записать данные, то другие клиенты не должны ни читать ни писать эти данные пока первый клиент не закончит (write lock).

Блокировка может быть наложена явно или неявно. Если клиент не назначает блокировку, MySQL сервер неявно устанавливает необходимый тип блокировки на время выполнения выражения или транзакции. В случае выполнения оператора SELECT сервер установит READ LOCK, а в случае UPDATE — WRITE LOCK.

В СУБД используют протокол доступа к данным, позволяющий избежать проблемы параллелизма. Его суть заключается в следующем:

* транзакция, результатом действия которой на строку данных в таблице является ее извлечение, обязана наложить блокировку чтения на эту строку;
* транзакция, предназначенная для модификации строки данных, накладывает на нее блокировку записи;
* если запрашиваемая блокировка на строку отвергается из-за уже имеющейся блокировки, то транзакция переводится в режим ожидания до тех пор, пока блокировка не будет снята;
* блокировка записи сохраняется вплоть до конца выполнения транзакции.

Решение проблемы параллельной обработки БД заключается в том, что строки таблиц блокируются, а последующие транзакции, модифицирующие эти строки, отвергаются и переводятся в режим ожидания. В связи со свойством сохранения целостности БД транзакции являются подходящими единицами изолированности пользователей. Действительно, если каждый сеанс взаимодействия с базой данных реализуется транзакцией, то пользователь начинает с того, что обращается к согласованному состоянию базы данных – состоянию, в котором она могла бы находиться, даже если бы пользователь работал с ней в одиночку.

Если в системе управления базами данных не реализованы механизмы блокирования, то при одновременном чтении и изменении одних и тех же данных несколькими пользователями могут возникнуть уже рассмотренные проблемы одновременного доступа.

### Взаимоблокировки

Большинство способов обеспечения параллелизма, хотя бы отчасти основанных на блокировках, подвержено взаимоблокировкам (deadlock). И хотя известны достаточно остроумные алгоритмы, позволяющие не допускать подобных ситуаций в принципе, в коммерческих приложениях они почти не встречаются. Microsoft SQL Server здесь не является исключением, и также подвержен взаимоблокировкам (они же «мертвые блокировки» или «тупиковые ситуации»).

Взаимоблокировка, как можно понять из названия – это ситуация, когда транзакции блокируют друг друга таким образом, что дальнейшее выполнение невозможно. В силу протокола двухфазной блокировки ни одна из участвующих во взаимоблокировке транзакций не может отпустить уже захваченные ей ресурсы до того, как наложит блокировки на все, что ей необходимо. А получить все необходимые ресурсы мешают уже наложенные блокировки. Таким образом, получается замкнутый круг. Естественно, и транзакций, и объектов в общем случае может быть сколь угодно много. Разорвать такую блокировку без внешнего вмешательства невозможно, и если не предпринимать специальных усилий, то транзакции будут находиться в состоянии ожидания бесконечно долго. Разрешить подобную ситуацию можно лишь путем отмены хотя бы одной из транзакций.

В Microsoft SQL Server используется механизм устранения взаимоблокировок на основе графа ожидания. Граф строится при каждом запросе блокировки. По истечении некоего тайм-аута просыпается монитор блокировок, и если он обнаруживает, что какая-то транзакция ждет слишком долго, инициируется процесс нахождения замкнутого цикла в графе ожидания. В случае обнаружения мертвой блокировки происходит откат одной из транзакций, участвующих в цикле. «Жертва» вычисляется в зависимости от объема проделанной работы, которая в свою очередь определяется по количеству записей в журнале транзакций, которые необходимо откатить. Однако есть возможность указать серверу, какую транзакцию предпочтительнее видеть в качестве «жертвы», с помощью команды:

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

Здесь @deadlock\_var – переменная в диапазоне от 1 до 12, чем меньше число, тем ниже приоритет; LOW соответствует 3, а NORMAL – 6.

Но как бы сервер не старался, все, что он сможет сделать по своей инициативе – это отменить одну из транзакций. Самостоятельно Microsoft SQL Server отмененную транзакцию заново не запускает, а возвращает сообщение об ошибке. Поэтому в клиентском приложении необходимо предусмотреть обработку данной ситуации и, возможно, перезапуск отмененной транзакции.

В свете всего вышеописанного почетной обязанностью разработчика является сведение вероятности мертвой блокировки к минимуму, а в идеале – к нулю, что является достаточно сложной, но вполне разрешимой задачей.

Строго говоря, все случаи взаимоблокировки сводятся к нарушению порядка доступа к объектам.

**Пример 1**

Самый тривиальный случай, нарушение порядка доступа в чистом виде, представляют собой две транзакции примерно следующего содержания:

Первая транзакция - T1.

BEGIN TRAN

UPDATE Tbl SET X = 1 WHERE X = 1

UPDATE Tbl SET X = 3 WHERE X = 3

COMMIT TRAN

Вторая транзакция - T2.

BEGIN TRAN

UPDATE Tbl SET X = 3 WHERE X = 3

UPDATE Tbl SET X = 1 WHERE X = 1

COMMIT TRAN

Если эти транзакции стартуют одновременно, то произойдет взаимоблокировка по причине очевидного нарушения порядка доступа. T1 сначала обращается к записи X = 1, а затем к записи X = 3. Т2 же, наоборот, сначала обращается к записи X = 3, а затем к X = 1.

подобная ситуация без грубого вмешательства неразрешима, то одна из транзакций будет отменена, другая же, пользуясь тем, что блокировка исчезла, спокойно завершит свою работу.

Решение: достаточно просто поменять порядок операторов в одной из транзакций. В общем случае необходимо добиться того, чтобы все транзакции обращались к объектам в одном и том же порядке.

**Пример 2:**

Очень часто встречается примерно такая последовательность операторов в одной транзакции:

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

BEGIN TRAN

SELECT @Var = Y FROM Tbl WHERE X = 2

---

--- здесь выполняются какие-нибудь вычисления над @Var.

---

UPDATE Tbl SET Y = @Var WHERE X = 2

COMMIT TRAN

Если два экземпляра такой транзакции запустить одновременно из двух разных потоков, то с очень большой вероятностью все закончится взаимоблокировкой.

Выполним вышеприведенный T-SQL-код из транзакций T1 и T2. Чтобы имитировать возможное развитие событий при параллельной работе будем выполнять транзакции по частям. Сначала половину T1, затем целиком T2, а потом оставшуюся часть T1. Эффект будет точно таким же, как если бы в реальном приложении между двумя операторами T1 успела бы пролезть транзакция T2. На самом деле для получения взаимоблокировки достаточно, чтобы между двумя операторами T1 успел втиснуться только первый оператор T2, дальнейший порядок операций уже не важен.

Итак, выполним первую часть T1 в одном из окон Query Analyser’а:

--- установим необходимый уровень изоляции

SET ISOLATION LEVEL REPEATABLE READ

BEGIN TRANSACTION

SELECT \* FROM Tbl WHERE X = 2

Все выполнилось успешно, но транзакция все еще считается активной, мы ее не отменили и не зафиксировали. Откроем новое соединение с той же базой в новом окне QA и попытаемся выполнить эту же транзакцию целиком:

--- установим необходимый уровень изоляции

SET ISOLATION LEVEL REPEATABLE READ

BEGIN TRAN

SELECT \* FROM Tbl WHERE X = 2

UPDATE Tbl SET Y = 3 WHERE X = 2

COMMIT TRAN

SELECT выполнился, но до UPDATE дело не дошло, T2 ждет, пока T1 освободит запись X=2, чтобы наложить эксклюзивную блокировку.

Переключимся обратно в первое окно и попытаемся завершить T1:

UPDATE Tbl SET Y=3 WHERE X=2

COMMIT TRAN

Теперь и T1 будет ждать, пока T2 освободит свою коллективную блокировку. Таким образом, транзакции будут ожидать друг друга, цикл в графе ожидания замкнется и, некоторое время спустя, когда менеджер блокировок это обнаружит, одна из транзакций будет отменена. Приложение, запустившее ее, получит сообщение 1205 о взаимоблокировке (*Transaction (Process ID 61) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction)*, а другая транзакция завершится успешно.

В сущности, мы здесь опять имеем дело с нарушением порядка доступа, но в иной форме. Сначала транзакция наложила слабую блокировку на ресурс, а потом вернулась к этому же ресурсу, чтобы наложить более сильную. Особенность этой достаточно часто встречающейся ситуации в том, что это не две разных транзакции, а одна и та же, просто запущенная из разных сессий.

Решение:

Поскольку взаимоблокировка произошла из-за того, что транзакции удерживали коллективные блокировки и потом попытались их повысить до эксклюзивных, то, в принципе, помочь избежать неприятностей в данном случае сможет понижение уровня изоляции до READ COMMITED. При этом коллективная блокировка не будет держаться до конца транзакции, а снимется сразу после завершения чтения, а значит, обновить записи ничто не помешает. Но тогда вместо взаимоблокировки мы вполне можем получить неверные данные, так как между SELECT и UPDATE сможет втиснуться другая транзакция, которая изменит Y и данные, полученные SELECT’ на момент UPDATE, окажутся неактуальными, чего в некоторых случаях допускать нельзя.

Можно также сразу при чтении наложить эксклюзивную блокировку, но это тоже не самый лучший выход с точки зрения производительности, так как могут существовать транзакции, которым эти данные надо просто прочитать, а наложение эксклюзивной блокировки увеличивает время их пассивного ожидания.

В общем случае наилучшим выходом здесь будет наложение при чтении промежуточной блокировки обновления. Такая блокировка совместима с коллективной, что позволит читающим транзакциям обращаться к этим данным беспрепятственно. А когда понадобится их обновить, то проблем быть не должно, так как блокировки обновления между собой несовместимы, и значит, другие транзакции, читающие эти данные для последующего изменения (и естественно тоже запросившие их с блокировкой обновления), будут ждать, пока эти данные поменяются, никому не мешая. Для этого необходимо изменить первый оператор транзакции примерно таким образом:

SELECT @Var = Y FROM Tbl WITH (UPDLOCK) WHERE X = 2

### Блокировки в среде MS SQL Server

#### Управление блокировками

Пользователю чаще всего не нужно предпринимать никаких действий по управлению блокировками. Всю работу по установке, снятию и разрешению конфликтов выполняет специальный компонент сервера, называемый менеджером блокировок. MS SQL Server поддерживает различные уровни блокирования объектов (или детализацию блокировок), начиная с отдельной строки таблицы и заканчивая базой данных в целом. Менеджер блокировок автоматически оценивает, какое количество данных необходимо блокировать, и устанавливает соответствующий тип блокировки. Это позволяет поддерживать равновесие между производительностью работы системы блокирования и возможностью пользователей получать доступ к данным. Блокирование на уровне строки позволяет наиболее точно управлять таким доступом, поскольку блокируются только действительно изменяемые строки. Множество пользователей могут одновременно работать с данными с минимальными задержками. Платой за это является увеличение числа операций установки и снятия блокировок, а также большое количество служебной информации, которое приходится хранить для отслеживания установленных блокировок. При блокировке на уровне таблицы производительность системы блокирования резко увеличивается, так как необходимо установить лишь одну блокировку и снять ее только после завершения транзакции. Пользователь при этом имеет максимальную скорость доступа к данным. В то же время они не доступны никому другому, потому что вся таблица заблокирована. Приходится ожидать, пока текущий пользователь завершит работу.

Действия, выполняемые пользователями при работе с данными, сводятся к операциям двух типов: их чтению и изменению. В операции по изменению включаются действия по добавлению, удалению и собственно изменению данных. В зависимости от выполняемых действий сервер накладывает определенный тип блокировки из следующего перечня:

SQL Server поддерживает 4 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, 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.

**Блокировка массивного обновления**. Данный тип блокировок устанавливается сервером базы данных при выполнении массивного копирования данных в таблицы. Она запрещает обращение к таблице всем другим пользователям. Но, в тоже время, несколько запросов массированного копирования, могут одновременно добавлять строки в таблице.

Помимо перечисленных основных типов блокировок SQL Server поддерживает ряд специальных блокировок, предназначенных для повышения производительности и функциональности обработки данных. Они называются блокировками намерений и используются сервером в том случае, если транзакция намеревается получить доступ к данным вниз по иерархии и для других транзакций необходимо установить запрет на наложение блокировок, которые будут конфликтовать с блокировкой, накладываемой первой транзакцией.

Ранее рассмотренные блокировки относятся к данным. Помимо перечисленных в среде SQL Server существует два других типа блокировок: блокировка диапазона ключей и блокировка схемы (метаданных, описывающих структуру объекта).

Блокировка диапазона ключей решает проблему возникновения фантомов и обеспечивает требования сериализуемости транзакции. Блокировки этого типа устанавливаются на диапазон строк, соответствующих определенному логическому условию, с помощью которого осуществляется выборка данных из таблицы.

Блокировка схемы используется при выполнении команд модификации структуры таблиц для обеспечения целостности данных.

#### "Мертвые" блокировки

"Мертвые", или тупиковые, блокировки характерны для многопользовательских систем. "Мертвая" блокировка возникает, когда две транзакции блокируют два блока данных и для завершения любой из них нужен доступ к данным, заблокированным ранее другой транзакцией. Для завершения каждой транзакции необходимо дождаться, пока блокированная другой транзакцией часть данных будет разблокирована. Но это невозможно, так как вторая транзакция ожидает разблокирования ресурсов, используемых первой.

Без применения специальных механизмов обнаружения и снятия "мертвых" блокировок нормальная работа транзакций будет нарушена. Если в системе установлен бесконечный период ожидания завершения транзакции (а это задано по умолчанию), то при возникновении "мертвой" блокировки для двух транзакций вполне возможно, что, ожидая освобождения заблокированных ресурсов, в тупике окажутся и новые транзакции. Чтобы избежать подобных проблем, в среде MS SQL Server реализован специальный механизм разрешения конфликтов тупикового блокирования.

Для этих целей сервер снимает одну из блокировок, вызвавших конфликт, и откатывает инициализировавшую ее транзакцию. При выборе блокировки, которой необходимо пожертвовать, сервер исходит из соображений минимальной стоимости.

Полностью избежать возникновения "мертвых" блокировок нельзя. Хотя сервер и имеет эффективные механизмы снятия таких блокировок, все же при написании приложений следует учитывать вероятность их возникновения и предпринимать все возможные действия для предупреждения этого. "Мертвые" блокировки могут существенно снизить производительность, поскольку системе требуется достаточно много времени для их обнаружения, отката транзакции и повторного ее выполнения.

Для минимизации возможности образования "мертвых" блокировок при разработке кода транзакции следует придерживаться следующих правил:

* выполнять действия по обработке данных в постоянном порядке, чтобы не создавать условия для захвата одних и тех же данных;
* избегать взаимодействия с пользователем в теле транзакции;
* минимизировать длительность транзакции и выполнять ее по возможности в одном пакете;
* применять как можно более низкий уровень изоляции.

**Оптимистичные и пессимистичные блокировки**

Существует два вида блокировок - оптимистичная и пессимистичная.

**Оптимистичная блокировка**

При оптимистичной блокировке на базе данных реальной блокировки не происходит. Вместо этого используется следующий подход - если во время выполнения транзакции она изменяет данные, которые были изменены после её начала, то транзакция прерывается с исключением. Использование оптимистичных блокировок позволяет избежать взаимных блокировок (dead-lock). Для реализации оптимистичной блокировки часто используется версионирование данных - в таблицу добавляется колонка, которая хранит текущую версию. При выполнении update в запросе в секции where передается версия данных, которая была забрана на изменение. Если update вернул 0 изменённых строк, значит данные были уже изменены и транзакцию необходимо запускать заново. Вместо версии можно хранить время последнего изменения данных.

**Пессимистичная блокировка**

При пессимистичной блокировке для записи ставится эксклюзивная блокировка на уровне базы данных, запрещая таким образом доступ к данным из других транзакций. Существует несколько видов пессимистичных блокировок:

1. блокировка при чтении
2. блокировка при записи

При **блокировке при чтении** запись блокируется когда она запрашивается из базы данных. Недостаток метода в том, что таким образом можно заблокировать даже те данные, которые не изменяются в рамках текущей транзакции. При **блокировке при записи** блокировка даных происходит при их обновлении в базе данных до конца текущей транзакции. Блокировка с данных снимается либо при коммите, либо при откате транзакции.

**Сравнение оптимистичных и пессимистичных блокировок**

При разработке программного обеспечения необходимо выбирать стратегию блокировок данных. При этом следует учитывать следующее:

1. Если ситуация обновления одних и тех же данных в один момент времени относительно редка, то выгоднее использовать оптимистичную блокировку. В этом случае не будут происходить дорогая операция блокировки ресурсов.
2. Если же возможность возникновения ситуации обновления одних и тех же данных достаточно высока, то лучше использовать пессимистичную блокировку, это снизит количество прерванных транзакций.
3. Также следует учитывать, что при оптимистичной блокировке в случае прерывания транзакции её нужно запускать заново.

Ресурсы:

<https://en.wikipedia.org/wiki/Isolation_%28database_systems%29>

<https://ru.wikipedia.org/wiki/ACID>

<http://rsdn.ru/article/db/deadlocks.xml>

<http://shurshun.ru/tranzaktsii-blokirovki-urovni-izoliovannosti-tranzaktsiy-v-mysql/>

<http://datasql.ru/basesql/16.htm>

<http://atamanenko.blogspot.ru/2009/04/blog-post_22.html>

<http://rsdn.ru/article/db/mssqllocks.xml>