# 12. Модели транзакций

Оглавление



## 12.1. Понятие транзакции

Транзакцией называется последовательность операций, производимых над базой данных и переводящих базу данных из одного непротиворечивого (согласованного) состояния в другое непротиворечивое (согласованное) состояние.

Транзакция рассматривается как некоторое неделимое действие над базой данных, осмысленное с точки зрения пользователя. В то же время — это логическая единица работы системы. Рассмотрим несколько примеров. Что может быть названо транзакцией? Кем определяется, какая последовательность операций над базой данных составляет транзакцию? Конечно, однозначно именно разработчик определяет, какая последовательность операций составляет единое целое, то есть транзакцию. Разработчик приложений или хранимых процедур определяет это исходя из смысла обработки данных, именно семантика совокупности операций над базой данных, которая моделирует с точки зрения разработчика некоторую одну неразрывную работу, и составляет транзакцию. Допустим, выделим работу по вводу данных о поступивших книгах, новых книгах, которых не было раньше в библиотеке. Тогда эту операцию можно разбить на две последовательные: сначала ввод данных о книге — это новая строка в таблице BOOKS, а потом ввод данных обо всех экземплярах новой книги — это ввод набора новых строк в таблицу EXEMPLAR в количестве, равном количеству поступивших экземпляров

книги. Если эта последовательность работ будет прервана, то наша база данных не будет соответствовать реальному объекту, поэтому желательно выполнять ее как единую работу над базой данных.

Следующий пример, который связан с принятием заказа в фирме на изготовление компьютера. Компьютер состоит из комплектующих, которые сразу резервируются за данным заказом в момент его формирования. Тогда транзакцией будет вся последовательность операций, включающая следующие операции:

- ввод нового заказа со всеми реквизитами заказчика;
- изменения состояния для всех выбранных комплектующих на складе на "занято" с привязкой их к определенному заказу;
- подсчет стоимости заказа с формированием платежного документа типа выставляемого счета к оплате;
- включение нового заказа в производство.

С точки зрения работника, это единая последовательность операций; если она будет прервана, то база данных потеряет свое целостное состояние.

Еще один пример, который весьма характерен для учебных заведений. При длительной болезни преподавателя или при его увольнении перед администрацией кафедры встает задача перераспределения всей нагрузки, которую ведет преподаватель, по другим преподавателям кафедры. Видов нагрузки может быть несколько: чтение лекций и проведение занятий по текущему расписанию, руководство квалификационными работами бакалавров, руководство дипломными проектами специалистов, руководство магистерскими диссертациями, индивидуальная научно-исследовательская работа со студентами. И для каждого вида нагрузки необходимо найти исполнителей и назначить им дополнительную нагрузку.

## 12.2. Свойства транзакций. Способы завершения транзакций

Существуют различные модели транзакций, которые могут быть классифицированы на основании различных свойств, включающих структуру транзакции, параллельность внутри транзакции, продолжительность и т. д.

В настоящий момент выделяют следующие типы транзакций:

- плоские или классические транзакции,
- цепочечные транзакции и
- вложенные транзакции.

Плоские, или традиционные, транзакции, характеризуются четырьмя классическими свойствами: атомарности, согласованности, изолированности, долговечности (прочности) — *ACID* (Atomicity, Consistency, Isolation, Durability). Иногда традиционные транзакции называют *ACID-транзакциями*. Упомянутые выше свойства означают следующее:

- *Свойство атомарности* (Atomicity) выражается в том, что транзакция должна быть выполнена в целом или не выполнена вовсе.
- *Свойство согласованности* (Consistency) гарантирует, что по мере выполнения транзакций данные переходят из одного согласованного состояния в другое — транзакция не разрушает взаимной согласованности данных.
- *Свойство изолированности* (*Isolation*) означает, что конкурирующие за доступ к базе данных транзакции физически обрабатываются последовательно, изолированно друг от друга, но для пользователей это выглядит так, как будто они выполняются параллельно.
- *Свойство долговечности* (*Durability*) трактуется следующим образом: если транзакция завершена успешно, то те изменения в данных, которые были ею произведены, не могут быть потеряны ни при каких обстоятельствах (даже в случае последующих ошибок).

Возможны два варианта завершения транзакции. Если все *операторы* выполнены успешно и в процессе выполнения транзакции не произошло никаких сбоев программного или аппаратного обеспечения, *транзакция* фиксируется.

*Фиксация транзакции* — это действие, обеспечивающее *запись* на *диск* изменений в базе данных, которые были сделаны в процессе выполнения транзакции.

До тех пор, пока *транзакция* не зафиксирована, допустимо *аннулирование* этих изменений, восстановление *базы данных* в то состояние, в котором она была на момент начала транзакции. *Фиксация транзакции* означает, что все результаты выполнения транзакции становятся постоянными. Они станут видимыми другим транзакциям только после того, как текущая *транзакция* будет зафиксирована. До этого момента все данные, затрагиваемые транзакцией, будут "видны" пользователю в состоянии на начало текущей транзакции.

Если в процессе выполнения транзакции случилось нечто такое, что делает невозможным ее нормальное завершение, *база данных* должна быть возвращена в исходное состояние. *Откат* транзакции — это действие, обеспечивающее *аннулирование* всех изменений данных, которые были сделаны операторами *SQL* в теле текущей незавершенной транзакции.

Каждый оператор в транзакции выполняет свою часть работы, но для успешного завершения всей работы в целом требуется безусловное завершение всех их операторов. Группирование операторов в транзакции сообщает *СУБД*, что вся эта *группа* должна быть выполнена как единое целое, причем такое выполнение должно поддерживаться автоматически.

В стандарте ANSI/ISO *SQL* определены модель транзакций и функции операторов COMMIT и ROLLBACK. Стандарт определяет, что *транзакция* начинается с первого *SQL*-оператора, инициируемого пользователем или

содержащегося в программе, изменяющего текущее состояние *базы данных*. Все последующие *SQL-операторы* составляют тело транзакции. *Транзакция* завершается одним из четырех возможных путей (рис. 12.1):

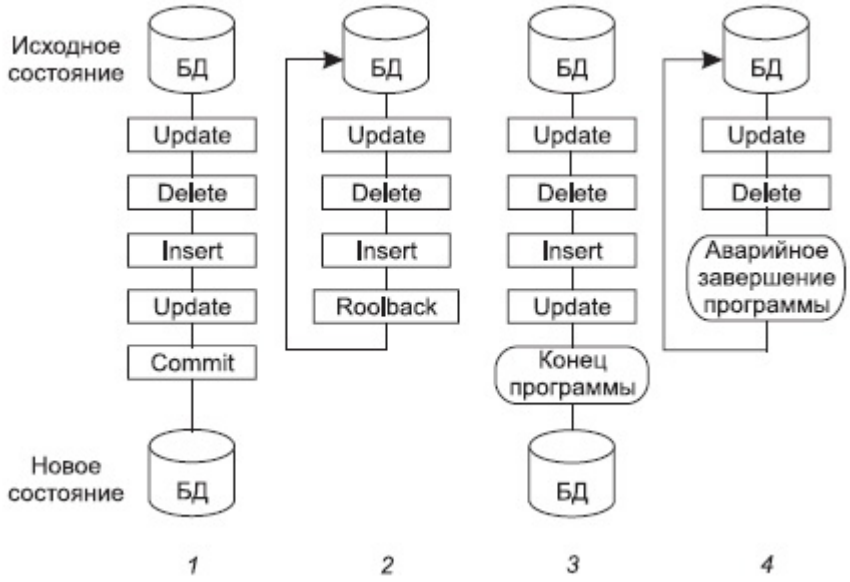


**Рис. 12.1.** Модель транзакций ANSI/ISO

1. оператор COMMIT означает успешное завершение транзакции; его использование делает постоянными изменения, внесенные в базу данных в рамках текущей транзакции;

2. оператор ROLLBACK прерывает транзакцию, отменяя изменения, сделанные в базе данных в рамках этой транзакции; новая транзакция начинается непосредственно после использования ROLLBACK ;

3. успешное завершение программы, в которой была инициирована текущая транзакция, означает успешное завершение транзакции (как будто был использован оператор COMMIT );

4. ошибочное завершение программы прерывает транзакцию (как будто был использован оператор ROLLBACK ).

В этой модели каждый оператор, который изменяет состояние *БД*, рассматривается как *транзакция*, поэтому при успешном завершении этого оператора *БД* переходит в новое устойчивое состояние.

В первых версиях коммерческих *СУБД* была реализована модель транзакций ANSI/ISO. В дальнейшем в *СУБД* SYBASE была реализована *расширенная модель* транзакций, которая включает еще ряд дополнительных операций. В модели SYBASE используются следующие четыре оператора:

- Оператор BEGIN TRANSACTION сообщает о начале транзакции. В отличие от модели в стандарте ANSI/ISO, где начало транзакции неявно задается первым оператором модификации данных, в модели SYBASE начало транзакции задается явно с помощью оператора начала транзакции.

- Оператор COMMIT TRANSACTION сообщает об успешном завершении транзакции. Он эквивалентен оператору COMMIT в модели стандарта ANSI/ISO. Этот оператор, как и оператор COMMIT, фиксирует все изменения, которые производились в БД в процессе выполнения транзакции.

- Оператор SAVE TRANSACTION создает внутри транзакции точку сохранения, которая соответствует промежуточному состоянию БД, сохраненному на момент выполнения этого оператора. В операторе SAVE TRANSACTION может стоять имя точки сохранения. Поэтому в ходе выполнения транзакции может быть запомнено несколько точек сохранения, соответствующих нескольким промежуточным состояниям.

- Оператор ROLLBACK имеет две модификации. Если этот оператор используется без дополнительного параметра, то он интерпретируется как оператор отката всей транзакции, то есть в этом случае он эквивалентен оператору отката ROLLBACK в модели ANSI/ISO. Если же оператор отката имеет параметр и записан в виде ROLLBACK B, то он интерпретируется как оператор частичного отката транзакции в точку сохранения B.

Принципы выполнения транзакций в расширенной модели транзакций представлены на рис. 12.2. На рисунке *операторы* помечены номерами, чтобы нам удобнее было проследить ход выполнения транзакции во всех допустимых случаях.

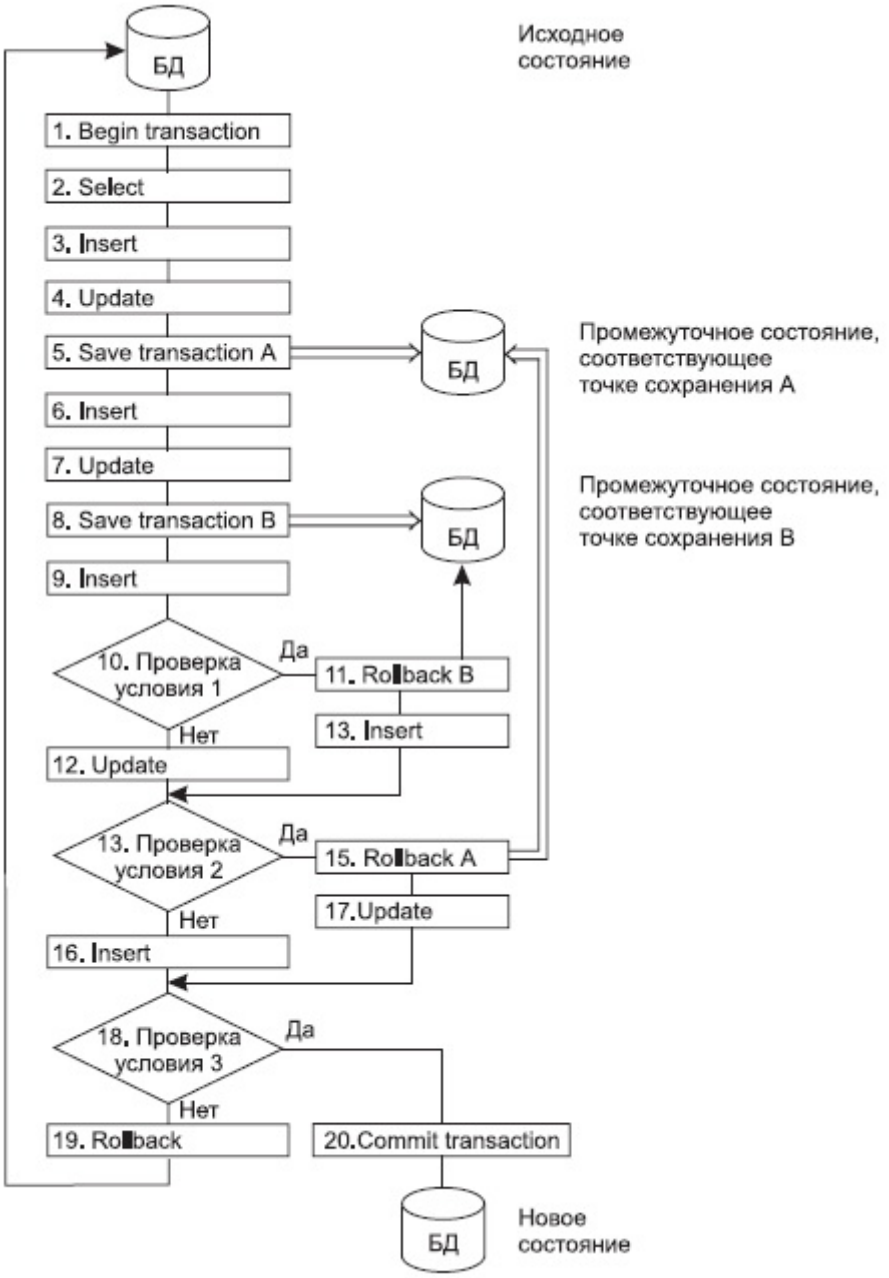


**Рис. 12.2.** Примеры выполнения транзакций в расширенной модели

*Транзакция* начинается явным оператором начала транзакции, который имеет в нашей схеме номер 1. Далее идет оператор 2, который является оператором поиска и не меняет текущее состояние *БД*, а следующие за ним *операторы* 3 и 4 переводят базу данных уже в новое состояние. Оператор 5 сохраняет это новое промежуточное состояние *БД* и помечает его как промежуточное состояние в точке А. Далее следуют *операторы* 6 и 7, которые переводят базу данных в новое состояние. А оператор 8 сохраняет это состояние как промежуточное состояние в точке В. Оператор 9 выполняет ввод новых данных, а оператор 10 проводит некоторую проверку условия 1; если условие 1 выполнено, то выполняется оператор 11, который проводит *откат* транзакции в промежуточное состояние В. Это означает, что последствия действий оператора 9 как бы стираются и *база данных* снова возвращается в промежуточное

состояние B, хотя после выполнения оператора 9 она уже находилась в новом состоянии. И после отката транзакции вместо оператора 9, который выполнялся раньше из состояния B *БД*, выполняется оператор 13 ввода новых данных, и далее управление передается оператору 14. Оператор 14 снова проверяет условие, но уже некоторое новое условие 2; если условие выполнено, то управление передается оператору 15, который выполняет *откат* транзакции в промежуточное состояние A, то есть все *операторы*, которые изменяли *БД*, начиная с 6 и заканчивая 13, считаются невыполненными, то есть результаты их выполнения исчезли и мы снова находимся в состоянии A, как после выполнения оператора 4. Далее управление передается оператору 17, который обновляет содержимое *БД*, после этого управление передается оператору 18, который связан с проверкой условия 3. Проверка заканчивается либо передачей управления оператору 20, который фиксирует транзакцию, и *БД* переходит в новое устойчивое состояние, и изменить его в рамках текущей транзакции невозможно. Либо, если управление передано оператору 19, то *транзакция* откатывается к началу и *БД* возвращается в свое начальное состояние, а все промежуточные состояния здесь уже проверены, и выполнить операцию отката в эти промежуточные состояния после выполнения оператора 19 невозможно.

Конечно, *расширенная модель* транзакции, предложенная фирмой SYBASE, поддерживает гораздо более гибкий механизм выполнения транзакций. Точки сохранения позволяют устанавливать маркеры внутри транзакции таким образом, чтобы имелась возможность отмены только части работы, проделанной в транзакции. Целесообразно использовать точки сохранения в длинных и сложных транзакциях, чтобы обеспечить возможность *отмены изменения* для определенных операторов. Однако это обусловливает дополнительные *затраты* ресурсов системы — оператор выполняет работу, а изменения затем отменяются; обычно усовершенствования в логике обработки могут оказаться более оптимальным решением.

## 12.3. Журнал транзакций

Реализация в *СУБД* принципа сохранения промежуточных состояний, подтверждения или отката транзакции обеспечивается специальным механизмом, для поддержки которого создается некоторая системная структура, называемая *Журналом транзакций*.

Однако назначение журнала транзакций гораздо шире. Он предназначен для обеспечения надежного хранения данных в *БД*.

А это требование предполагает, в частности, возможность восстановления согласованного состояния *базы данных* после любого рода аппаратных и программных сбоев. Очевидно, что для выполнения восстановлений необходима некоторая дополнительная *информация*. В подавляющем большинстве современных реляционных *СУБД* такая избыточная дополнительная *информация* поддерживается в виде *журнала изменений базы данных*, чаще всего называемого *Журналом транзакций*.

Итак, общей целью журнализации изменений баз данных является обеспечение возможности восстановления согласованного состояния *базы данных* после любого сбоя. Поскольку основой поддержания *целостного состояния базы данных* является механизм транзакций, *журнализация* и восстановление тесно связаны с понятием транзакции. Общими принципами восстановления являются следующие:

- результаты зафиксированных транзакций должны быть сохранены в восстановленном состоянии базы данных;
- результаты незафиксированных транзакций должны отсутствовать в восстановленном состоянии базы данных.

Это, собственно, и означает, что восстанавливается последнее по времени согласованное состояние *базы данных*.

Возможны следующие ситуации, при которых требуется производить восстановление состояния *базы данных*.

- **Индивидуальный откат** транзакции. Этот откат должен быть применен в следующих случаях:
  - стандартной ситуацией отката транзакции является ее явное завершение оператором ROLLBACK ;
  - аварийное завершение работы прикладной программы, которое логически эквивалентно выполнению оператора ROLLBACK, но физически имеет иной механизм выполнения;
  - принудительный откат транзакции в случае взаимной блокировки при параллельном выполнении транзакций. В подобном случае для выхода из тупика данная транзакция может быть выбрана в качестве "жертвы" и принудительно прекращено ее выполнение ядром СУБД.
- Восстановление после внезапной **потери содержимого оперативной памяти** (мягкий сбой). Такая ситуация может возникнуть в следующих случаях:
  - при аварийном выключении электрического питания;
  - при возникновении неустранимого сбоя процессора (например, срабатывании контроля оперативной памяти) и т. д. Ситуация характеризуется потерей той части базы данных, которая к моменту сбоя содержалась в буферах оперативной памяти.
- Восстановление после **поломки основного внешнего носителя** базы данных (жесткий сбой). Эта ситуация при достаточно высокой надежности современных *устройств внешней памяти* может возникать сравнительно редко, но тем не менее СУБД должна быть в состоянии восстановить базу данных даже и в этом случае. Основой восстановления является архивная копия и *журнал изменений* базы данных.

Для восстановления согласованного состояния *базы данных* при индивидуальном откате транзакции нужно устранить последствия операторов модификации *базы данных*, которые выполнялись в этой транзакции. Для восстановления непротиворечивого состояния *БД* при мягком сбое необходимо восстановить содержимое *БД* по содержимому журналов транзакций, хранящихся на дисках. Для восстановления согласованного состояния *БД* при жестком сбое надо восстановить содержимое *БД* по архивным копиям и журналам транзакций, которые хранятся на неповрежденных внешних носителях.

Во всех трех случаях основой восстановления является избыточное *хранение данных*. Эти избыточные данные хранятся в журнале, содержащем последовательность записей об изменении *базы данных*.

Возможны два основных варианта ведения журнальной информации. В первом варианте для каждой транзакции поддерживается отдельный локальный *журнал изменений базы данных* этой транзакцией. Такие журналы называются локальными журналами. Они используются для индивидуальных откатов транзакций и могут поддерживаться в оперативной (правильнее сказать, в виртуальной) памяти. Кроме того, поддерживается общий *журнал изменений базы данных*, используемый для восстановления состояния *базы данных* после мягких и жестких сбоев.

Этот подход позволяет быстро выполнять индивидуальные откаты транзакций, но приводит к дублированию информации в локальных и общем журналах. Поэтому чаще используется второй вариант — поддержание только общего *журнала изменений базы данных*, который используется и при выполнении индивидуальных откатов. Далее мы рассматриваем именно этот вариант.

Общая структура журнала условно может быть представлена в виде некоторого по*следовательного файла*, в котором фиксируется каждое изменение *БД*, которое происходит в ходе выполнения транзакции. Все транзакции имеют свои внутренние номера, поэтому в едином журнале транзакций фиксируются все изменения, проводимые всеми транзакциями.

Каждая *запись* в журнале транзакций помечается номером транзакции, к которой она относится, и значениями атрибутов, которые она меняет. Кроме того, для каждой транзакции в журнале фиксируется *команда* начала и завершения транзакции (см. рис. 12.3).

Для большей надежности *журнал транзакций* часто дублируется системными средствами коммерческих *СУБД*, именно поэтому объем внешней памяти во много раз превышает реальный объем данных, которые хранятся в хранилище.

Имеются два альтернативных варианта ведения журнала транзакций: протокол с отложенными обновлениями и протокол с немедленными обновлениями.

Ведение журнала по принципу отложенных изменений предполагает следующий механизм выполнения транзакций:

1. Когда транзакция T1 начинается, в протокол заносится запись <T1 Begin transaction>

2. На протяжении выполнения транзакции в протоколе для каждой изменяемой записи записывается новое значение: <T1,ID_RECORD, атрибут, новое значение … >. Здесь ID_RECORD — уникальный номер записи.

3. Если все действия, из которых состоит транзакция T1, успешно выполнены, то транзакция частично фиксируется и в протокол заносится <T1 COMMIT>.

4. После того как транзакция фиксирована, записи протокола, относящиеся к T1, используются для внесения соответствующих изменений в БД.

5. Если происходит сбой, то СУБД просматривает протокол и выясняет, какие транзакции необходимо переделать. Транзакцию T1 необходимо переделать, если протокол содержит обе записи <T1 BEGIN TRANSACTION> и <T1 COMMIT>. БД может находиться в несогласованном состоянии, однако все новые значения измененных элементов данных содержатся в протоколе, и это требует повторного выполнения транзакции. Для этого используется некоторая системная процедура REDO(), которая заменяет все значения элементов данных на новые, просматривая протокол в прямом порядке.

6. Если в протоколе не содержится команда *фиксации транзакции* COMMIT, то никаких действий проводить не требуется, а транзакция запускается заново.

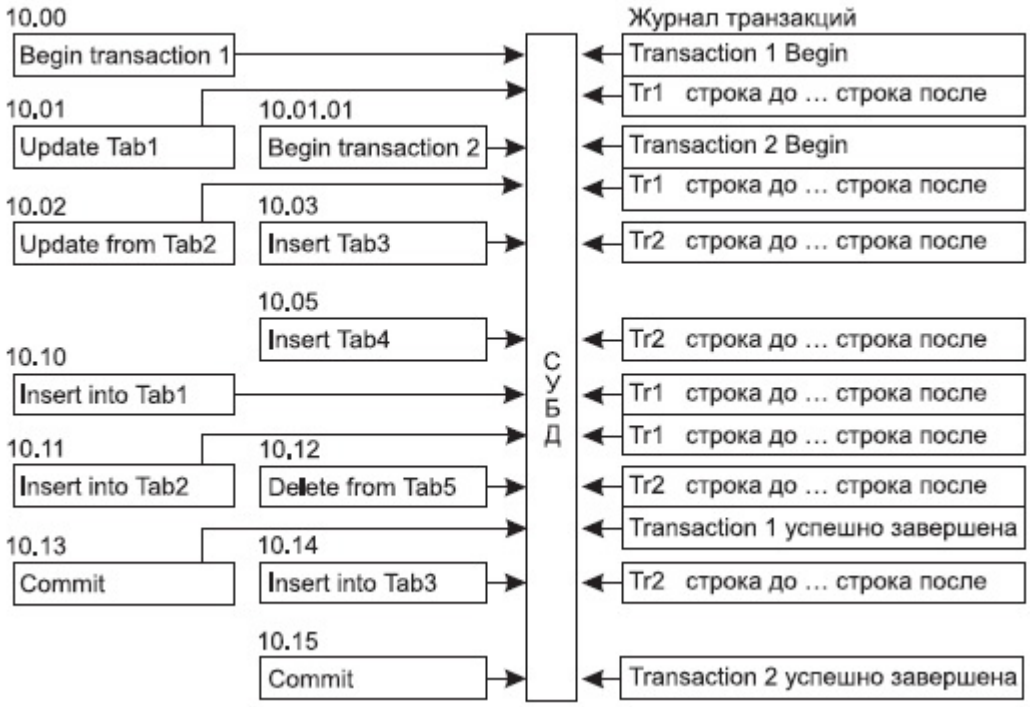

**Рис. 12.3.** Журнал транзакций

Альтернативный механизм с немедленным выполнением предусматривает внесение изменений сразу в *БД*, а в протокол заносятся не только новые, но и все старые значения изменяемых атрибутов, поэтому каждая *запись* выглядит <T1, IDRECORD, атрибут новое значение старое значение ...>. При этом *запись* в журнал предшествует непосредственному выполнению *операции* над *БД*. Когда *транзакция* фиксируется, то есть встречается *команда* <T1 COMMIT> и она выполняется, то все изменения оказываются уже внесенными в *БД* и не требуется никаких дальнейших действий по отношению к этой транзакции.

При откате транзакции выполняется системная процедура UNDO(), которая возвращает все старые значения в отмененной транзакции, последовательно проходя по протоколу начиная с команды BEGIN TRANSACTION.

Для восстановления при сбое используется следующий механизм:

- Если транзакция содержит команду начала транзакции, но не содержит команды фиксации с подтверждением ее выполнения, то выполняется последовательность действий как при откате транзакции, то есть восстанавливаются старые значения.

- Если сбой произошел после выполнения последней команды изменения БД, но до выполнения команды фиксации, то команда фиксации выполняется, а с БД никаких изменений не происходит. Работа происходит только на уровне протокола.

- Однако следует отметить, что проблемы восстановления выглядят гораздо сложнее приведенных ранее алгоритмов, с учетом того, что изменения как в журнал, так и в БД заносятся не сразу, а буферируются. Этому посвящен следующий раздел.

## 12.4. Журнализация и буферизация

*Журнализация* изменений тесно связана не только с управлением транзакциями, но и с буферизацией страниц *базы данных* в оперативной памяти.

Если бы *запись* об изменении *базы данных*, которая должна поступить в журнал при выполнении любой *операции* модификации *базы данных*, реально немедленно записывалась бы во внешнюю *память*, это привело бы к существенному замедлению работы системы. Поэтому записи в журнале тоже буферизуются: при нормальной работе очередная страница выталкивается во внешнюю *память* журнала только при полном заполнении записями.

Проблема состоит в выработке некоторой общей политики выталкивания, которая обеспечивала бы возможность восстановления состояния *базы данных* после сбоев.

Проблема не возникает при индивидуальных откатах транзакций, поскольку в этих случаях содержимое оперативной памяти не утрачено и можно пользоваться содержимым как буфера журнала, так и буферов страниц *базы данных*. Но если произошел мягкий сбой и содержимое буферов утрачено, для проведения восстановления *базы данных* необходимо иметь некоторое согласованное состояние журнала и *базы данных* во внешней памяти.

Основным принципом согласованной политики выталкивания буфера журнала и буферов страниц *базы данных* является то, что *запись* об изменении объекта *базы данных* должна попадать во внешнюю *память* журнала раньше, чем измененный *объект* оказывается во внешней памяти *базы данных*. Соответствующий протокол журнализации (и управления буферизацией) называется Write Ahead Log (WAL) — "пиши сначала в журнал" и состоит в том, что если требуется записать во внешнюю *память* измененный *объект базы данных*, то перед этим нужно гарантировать *запись* во внешнюю *память* журнала транзакций записи о его изменении.

Другими словами, если во внешней памяти *базы данных* находится некоторый *объект базы данных*, по отношению к которому выполнена операция модификации, то во внешней памяти журнала обязательно находится *запись*, соответствующая этой *операции*. Обратное неверно, то есть если во внешней памяти журнале содержится *запись* о некоторой *операции* изменения объекта *базы данных*, то сам измененный *объект* может отсутствовать во внешней памяти *базы данных*.

Дополнительное условие на выталкивание буферов накладывается тем требованием, что каждая успешно завершившаяся *транзакция* должна быть реально зафиксирована во внешней памяти. Какой бы сбой не произошел, система должна быть в состоянии восстановить состояние *базы данных*, содержащее результаты всех зафиксированных к моменту сбоя транзакций.

Простым решением было бы выталкивание буфера журнала, за которым следует массовое выталкивание буферов страниц *базы данных*, изменявшихся данной транзакцией. Довольно часто так и делают, но это вызывает существенные накладные *расходы* при выполнении *операции фиксации транзакции*.

Оказывается, что минимальным требованием, гарантирующим возможность восстановления последнего согласованного состояния *базы данных*, является выталкивание при *фиксации транзакции* во внешнюю *память* журнала всех записей об изменении *базы данных* этой транзакцией. При этом последней записью в журнал, производимой от имени данной транзакции, является специальная *запись* о конце транзакции.

Рассмотрим теперь, как можно выполнять *операции* восстановления *базы данных* в различных ситуациях, если в системе поддерживается общий для всех транзакций журнал с общей буферизацией записей, поддерживаемый в соответствии с протоколом WAL.

## 12.5. Индивидуальный откат транзакции

Для того чтобы можно было выполнить по общему журналу индивидуальный откат транзакции, все записи в журнале по данной транзакции связываются в обратный список. Началом списка для не закончившихся транзакций является запись о последнем изменении базы данных, произведенном данной транзакцией. Для закончившихся транзакций (индивидуальные откаты которых уже невозможны) началом списка является запись о конце транзакции, которая обязательно вытолкнута во внешнюю память журнала. Концом списка всегда служит первая запись об изменении базы данных, произведенном данной транзакцией. Обычно в каждой записи проставляется уникальный идентификатор транзакции, чтобы можно было восстановить прямой список записей об изменениях базы данных данной транзакцией.

Итак, индивидуальный откат транзакции (еще раз подчеркнем, что это возможно только для не закончившихся транзакций) выполняется следующим образом:

- Выбирается очередная запись из списка данной транзакции.
- Выполняется противоположная по смыслу операция: вместо операции INSERT выполняется соответствующая операция DELETE, вместо операции DELETE выполняется INSERT и вместо прямой операции UPDATE обратная операция UPDATE, восстанавливающая предыдущее состояние объекта базы данных.
- Любая из этих обратных операций также заносится в журнал. Собственно, для индивидуального отката это не нужно, но при выполнении индивидуального отката транзакции может произойти мягкий сбой, при восстановлении после которого потребуется откатить такую транзакцию, для которой не полностью выполнен индивидуальный откат.
- При успешном завершении отката в журнал заносится запись о конце транзакции. С точки зрения журнала такая транзакция является зафиксированной.

## 12.6. Восстановление после мягкого сбоя

К числу основных проблем восстановления после мягкого сбоя относится то, что одна логическая операция изменения базы данных может изменять несколько физических блоков базы данных, например, страницу данных и несколько страниц индексов. Страницы базы данных буферизуются в оперативной памяти и выталкиваются независимо. Несмотря на применение протокола WAL, после мягкого сбоя набор страниц внешней памяти базы данных может оказаться несогласованным, то есть часть страниц внешней памяти соответствует объекту до изменения, часть — после изменения. К такому состоянию объекта неприменимы операции логического уровня.

Состояние внешней памяти базы данных называется физически согласованным, если наборы страниц всех объектов согласованы, то есть соответствуют состоянию объекта либо до его изменения, либо после изменения.

Будем считать, что в журнале отмечаются точки физической согласованности базы данных — моменты времени, в которые во внешней памяти содержатся согласованные результаты операций, завершившихся до соответствующего момента времени, и *отсутствуют результаты* операций, которые не завершились, а буфер журнала вытолкнут во внешнюю память. Немного позже мы рассмотрим, как можно достичь физической согласованности. Назовем такие точки *tpc* (time of physical consistency) — точками физического согласования.

Тогда к моменту мягкого сбоя возможны следующие состояния транзакций:

- транзакция успешно завершена, то есть выполнена операция подтверждения транзакции COMMIT и для всех операций транзакции получено подтверждение ее выполнения во внешней памяти;
- транзакция успешно завершена, но для некоторых операций не получено подтверждение их выполнения во внешней памяти;
- транзакция получила и выполнила команду отката **ROLLBACK**;
- транзакция не завершена.

## 12.7. Физическая согласованность базы данных

Каким же образом можно обеспечить наличие точек физической согласованности базы данных, то есть как восстановить состояние базы данных в момент *tpc*? Для этого используются два основных подхода: подход, основанный на использовании теневого механизма, и подход, в котором применяется *журнализация* постраничных изменений базы данных.

При открытии файла таблица отображения номеров его логических блоков в адреса физических блоков внешней памяти считывается в оперативную память. При модификации любого блока файла во внешней памяти выделяется новый блок. При этом текущая таблица отображения (в оперативной памяти) изменяется, а теневая — сохраняется неизменной. Если во время работы с открытым файлом происходит сбой, во внешней памяти автоматически сохраняется состояние файла до его открытия. Для явного восстановления файла достаточно повторно считать в оперативную память теневую таблицу отображения.

Общая идея теневого механизма показана на рис. 12.4.

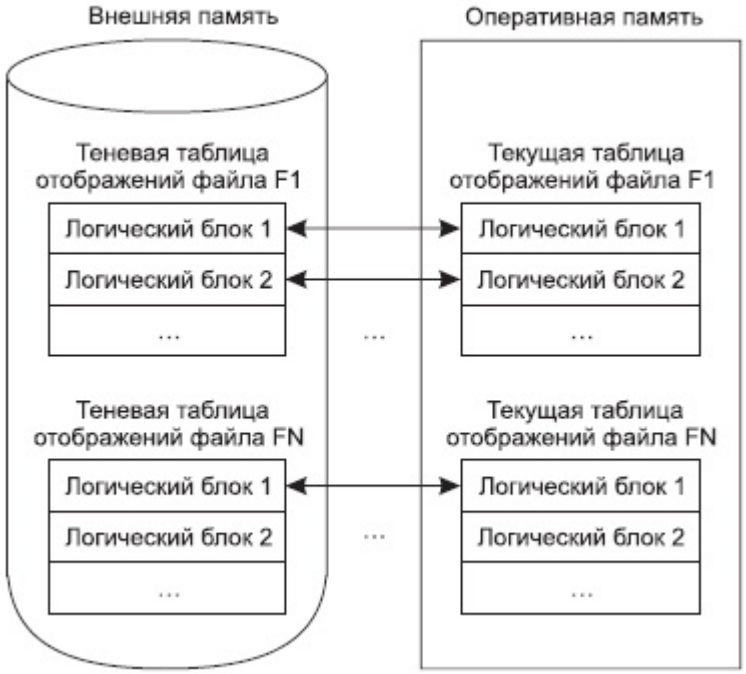


**Рис. 12.4.** Использование теневых таблиц отображения информации

В контексте базы данных теневой механизм используется следующим образом. Периодически выполняются операции установления точки физической согласованности базы данных (*checkpoints*). Для этого все логические операции завершаются, все буферы оперативной памяти, содержимое которых не соответствует содержимому соответствующих страниц внешней памяти, выталкиваются. Теневая таблица отображения файлов базы данных заменяется на текущую (правильнее сказать, текущая таблица отображения записывается на место теневой).

Восстановление к *tpc* происходит мгновенно: текущая таблица отображения заменяется на теневую (при восстановлении просто считывается теневая таблица отображения). Все проблемы восстановления решаются, но за счет слишком большого перерасхода внешней памяти. В пределе может потребоваться вдвое больше внешней памяти, чем реально нужно для хранения базы данных. Теневой механизм — это надежное, но слишком грубое средство. Обеспечивается согласованное состояние внешней памяти в один общий для всех объектов момент времени. На самом деле достаточно иметь совокупность согласованных наборов страниц, каждому из которых может соответствовать свои временные отсчеты.

Для выполнения такого более слабого требования наряду с логической журнализацией операций изменения базы данных производится *журнализация* постраничных изменений. Первый этап восстановления после мягкого сбоя состоит в постраничном откате незакончившихся логических операций. Подобно тому как это делается с логическими записями по отношению к транзакциям, последней записью о постраничных изменениях от одной логической операции является запись о конце операции.

В этом подходе имеются два метода решения проблемы. При использовании первого метода поддерживается общий журнал логических и страничных операций. Естественно, наличие двух видов записей, интерпретируемых абсолютно по-разному, усложняет структуру журнала. Кроме того, записи о постраничных изменениях, актуальность которых носит локальный характер, существенно (и не очень осмысленно) увеличивают журнал.

Поэтому все более популярным становится поддержание отдельного (короткого) журнала постраничных изменений. Такая техника применяется, например, в известном продукте Informix Online.

Предположим, что некоторым способом удалось восстановить внешнюю память базы данных к состоянию на момент времени *tpc* (как это можно сделать — немного позже). Тогда:

- Для транзакции T1 никаких действий производить не требуется. Она закончилась до момента *tpc*, и все ее результаты отражены во внешней памяти базы данных.
- Для транзакции T2 нужно повторно выполнить оставшуюся часть операций ( redo ). Действительно, во внешней памяти полностью отсутствуют следы операций, которые выполнялись в транзакции T2 после момента *tpc*. Следовательно, повторная прямая интерпретация операций T2 корректна и приведет к логически согласованному состоянию базы данных (поскольку транзакция T2 успешно завершилась до момента мягкого сбоя, в журнале содержатся записи обо всех изменениях, произведенных этой транзакцией).
- Для транзакции T3 нужно выполнить в обратном направлении первую часть операций ( undo ). Действительно, во внешней памяти базы данных полностью *отсутствуют результаты* операций T3, которые были выполнены после момента *tpc*. С другой стороны, во внешней памяти гарантированно присутствуют результаты операций T3, которые были выполнены до момента *tpc*. Следовательно, обратная интерпретация операций T3 корректна и приведет к согласованному состоянию базы данных (поскольку транзакция T3 не завершилась к моменту мягкого сбоя, при восстановлении необходимо устранить все последствия ее выполнения).
- Для транзакции T4, которая успела начаться после момента *tpc* и закончиться до момента мягкого сбоя, нужно выполнить полную повторную прямую интерпретацию операций ( redo ).
- Наконец, для начавшейся после момента *tpc* и не успевшей завершиться к моменту мягкого сбоя транзакции T5 никаких действий предпринимать не требуется. Результаты операций этой транзакции полностью отсутствуют во внешней памяти базы данных.

## 12.8. Восстановление после жесткого сбоя

Понятно, что для восстановления последнего согласованного состояния базы данных после жесткого сбоя *журнала изменений* базы данных явно недостаточно. Основой восстановления в этом случае являются журнал и архивная копия базы данных.

Восстановление начинается с обратного копирования базы данных из архивной копии. Затем для всех закончившихся транзакций выполняется redo, то есть операции повторно выполняются в прямом порядке.

Более точно, происходит следующее:

- по журналу в прямом направлении выполняются все операции;
- для транзакций, которые не закончились к моменту сбоя, выполняется откат.

На самом деле, поскольку жесткий сбой не сопровождается утратой буферов оперативной памяти, можно восстановить базу данных до такого уровня, чтобы можно было продолжить даже выполнение незакончившихся транзакций. Но обычно это не делается, потому что восстановление после жесткого сбоя — это достаточно длительный процесс.

Хотя к ведению журнала предъявляются особые требования по части надежности, в принципе возможна и его утрата. Тогда единственным способом восстановления базы данных является возврат к архивной копии. Конечно, в этом случае не удастся получить последнее согласованное состояние базы данных, но это лучше, чем ничего.

Последний вопрос, который мы коротко рассмотрим, относится к производству архивных копий базы данных. Самый простой способ — архивировать базу данных при переполнении журнала. В журнале вводится так называемая "желтая зона", при достижении которой образование новых транзакций временно блокируется. Когда все транзакции закончатся и, следовательно, база данных придет в согласованное состояние, можно производить ее архивацию, после чего начинать заполнять журнал заново.

Можно выполнять архивацию базы данных реже, чем переполняется журнал. При переполнении журнала и окончании всех начатых транзакций можно архивировать сам журнал. Поскольку такой архивированный журнал, по сути дела, требуется только для воссоздания архивной копии базы данных, журнальная информация при архивации может быть существенно сжата.

## 12.9. Параллельное выполнение транзакций

Если с *БД* работают одновременно несколько пользователей, то обработка транзакций должна рассматриваться с новой точки зрения. В этом случае *СУБД* должна не только корректно выполнять индивидуальные транзакции и восстанавливать согласованное состояние *БД* после сбоев, но она призвана обеспечить корректную параллельную работу всех пользователей над одними и теми же данными. По теории каждый по*льзователь* и каждая *тран-*

*закция* должны обладать свойством изолированности, то есть они должны выполняться так, как если бы только один по*льзователь* работал с *БД*. И средства современных *СУБД* позволяют изолировать пользователей друг от друга именно таким образом. Однако в этом случае возникают проблемы замедления работы пользователей. Рассмотрим более подробно проблемы, которые возникают при параллельной обработке транзакций.

Основные проблемы, которые возникают при параллельном выполнении транзакций, делятся условно на 4 типа:

- **Пропавшие изменения**. Эта ситуация может возникать, если две транзакции одновременно изменяют одну и ту же запись в БД. Например, работают два оператора на приеме заказов, первый оператор принял заказ на 30 мониторов. Когда он запрашивал склад, то там числилось 40 мониторов, и он, получив подтверждение от клиента, выставил счет и оформил продажу 30 мониторов из 40. Параллельно с ним работает второй оператор, который принимает заказ на 20 таких же мониторов (ну уж очень хорошая модель и дешево) и, в свою *очередь запросив* состояние склада и получив исходно ту же цифру 40, он успешно оформляет заказ для своего клиента. Заканчивая работу с данным заказом, он выполняет команду **Обновить** ( UPDATE ), которая заносит 20 как остаток любимых мониторов на складе. Но после этого, наконец, любезно попрощавшись со своим клиентом и заверив его в скорейшей доставке заказанных мониторов, заканчивает работу со своим заказом первый оператор и также выполняет команду **Обновить** и заносит 10 как остаток тех же мониторов на складе. Каждый из них доволен своей работой, но мы-то знаем, что произошло. Прежде всего, они продали 50 мониторов из наличествующих 40 штук, и далее на складе еще числится 10 подобных мониторов. БД теперь находится в несогласованном состоянии, а у фирмы возникли серьезные проблемы. Изменения, сделанные вторым оператором, были проигнорированы программой выполнения заказа, с которой работал первый оператор. Подобная ситуация представлена на рис. 12.5.

- **Проблемы промежуточных данных**. Рассмотрим ту же проблему одновременной работы двух операторов. Допустим, первый оператор, ведя переговоры со своим заказчиком, ввел заказанные 30 мониторов, но перед окончательным оформлением заказа клиент захотел выяснить еще некоторые характеристики товара. Приложение, с которым работает первый оператор, уже изменило остаток мониторов на складе, и там сейчас находится информация о 10 оставшихся мониторах. В это время второй оператор пытается принять заказ от своего клиента на 20 мониторов, но его приложение показывает, что на складе осталось всего 10 мониторов, и оператор вынужден отказать выгодному клиенту, который идет в другую фирму, весьма неудовлетворенный работой нашей компании. А в этот момент клиент оператора 1 заканчивает обсуждение дополнительных характеристик наших мониторов и принимает весьма невыгодное решение не покупать у нас мониторы, и приложение оператора 1 выполняет откат транзакции, и на складе снова оказывается 40 мониторов. Мы потеряли выгодного заказчика, но еще хуже было бы, если

бы клиент второго оператора согласился на 10 оставшихся мониторов, и приложение, с которым работает оператор два, отработав свой алгоритм, занесло 0 (ноль) оставшихся мониторов на складе, а после этого приложение оператора один снова бы записало исходные 40 мониторов на складе, хотя 10 из них уже проданы. Такая ситуация оказалась возможной потому, что приложение второго оператора имело доступ к промежуточным данным, которые сформировало первое приложение.

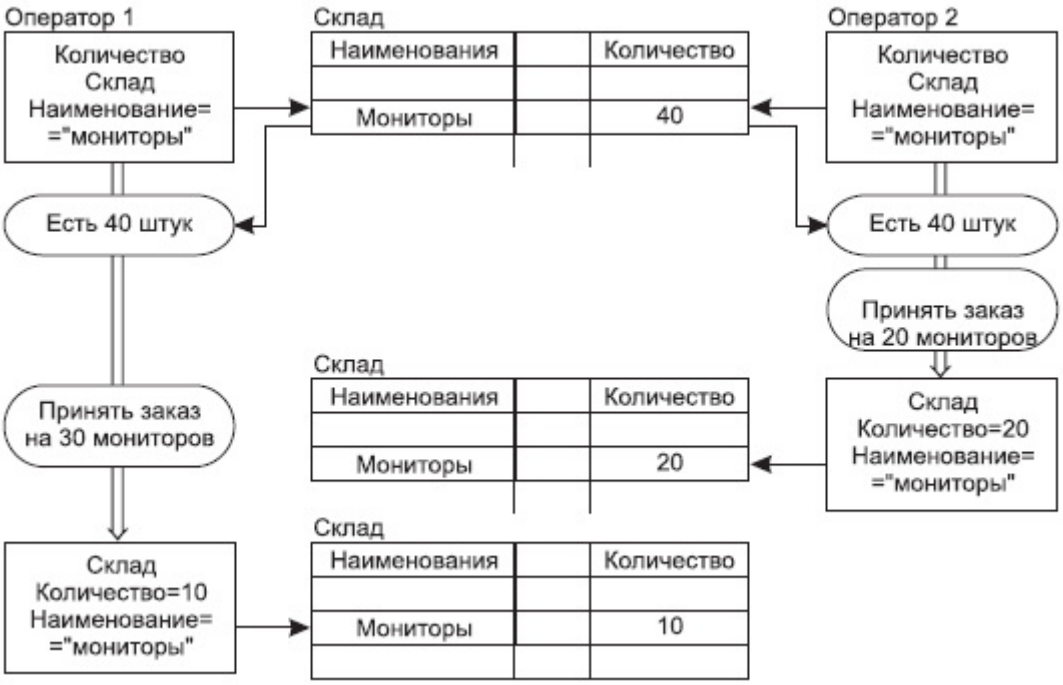


**Рис. 12.5.** Проблема пропавших обновлений

- **Проблемы несогласованных данных.** Рассмотрим ту же самую ситуацию с заказом мониторов. Предположим, что ситуация несколько изменилась. И оба оператора начинают работать практически одновременно. Они оба получают начальное состояние склада 40 мониторов, а далее первый оператор успешно завершает переговоры со своим клиентом и продает ему 30 мониторов. Он завершает работу своего приложения, и оно выполняет команду *фиксации транзакции* COMMIT. Состояние базы данных непротиворечивое. В этот момент, выяснив все тонкости и характеристики наших мониторов, клиент второго оператора также решает сделать заказ, и второй оператор, повторно получая состояние склада, видит, что оно изменилось. База данных находится в непротиворечивом состоянии, но второй оператор считает, что нарушена целостность его транзакции, в течение выполнения одной работы он получил два различных состояния склада. Эта ситуация возникла потому, что приложение первого оператора смогло изменить кортеж с данными, который уже прочитало приложение второго оператора.

- **Проблемы строк-призраков** (строк-фантомов). Предположим, что администратор нашей фирмы поручил секретарю напечатать итоговый отчет по результатам работы за текущий месяц. И допустим, что приложение

печатает отчет в двух видах: в подробном и в укрупненном. В момент, когда приложение печати начало формировать свой первый вид отчета, один из операторов принимает еще один заказ, поэтому к моменту формирования укрупненного отчета в БД появились новые сведения о продажах, которые и были внесены в укрупненный отчет. Мы получили два отчета в одном приложении, которые содержат разные цифры и не совпадают друг с другом. Такое стало возможно потому, что приложение печати выполнило два одинаковых запроса и получило два разных результата. БД находится в согласованном состоянии, но приложение печати работает некорректно.

Для того чтобы избежать подобных проблем, требуется выработать некоторую процедуру согласованного выполнения параллельных транзакций. Эта процедура должна удовлетворять следующим **правилам**:

- В ходе выполнения транзакции пользователь видит только согласованные данные. Пользователь не должен видеть несогласованных промежуточных данных.
- Когда в БД две транзакции выполняются параллельно, то СУБД гарантированно поддерживает принцип независимого выполнения транзакций, который гласит, что результаты выполнения транзакций будут такими же, как если бы вначале выполнялась транзакция 1, а потом транзакция 2, или наоборот, сначала транзакция 2, а потом транзакция 1.

Такая процедура называется сериализацией транзакций. Фактически она гарантирует, что каждый по*льзователь* (*программа*), обращающаяся к базе данных, работает с ней так, как будто не существует других пользователей (программ), одновременно с ним обращающихся к тем же данным.

Для поддержки параллельной работы транзакций строится специальный план.

План (способ) выполнения набора транзакций называется сериальным, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций.

Самым простым было бы последовательное выполнение транзакций, но такой план не оптимален по времени, существуют более гибкие методы управления параллельным доступом к *БД*. Наиболее распространенным механизмом, который используется коммерческими *СУБД* для реализации на практике сериализации транзакций является механизм блокировок. Самый простой вариант — это *блокировка* объекта на все время действия транзакции. Подобный пример рассмотрен на рис. 12.6. Здесь две транзакции, названные условно A и B, работают с тремя таблицами: T1, T2 и T3. В момент начала работы с любым объектом этот *объект* блокируется транзакцией, которая с ним начала работу, и он становится недоступным всем другим транзакциям до окончания транзакции, заблокировавшей ("захватившей") данный *объект*. После окончания транзакции все заблокированные ею объекты разблокируются и стано-

вятся доступными другим транзакциям. Если *транзакция* обращается к заблокированному объекту, то она остается в состоянии ожидания до момента разблокировки этого объекта, после чего она может продолжать обработку данного объекта. Поэтому *транзакция* ожидает разблокировки таблицы T2 транзакцией A. Над прямоугольниками стоит условное *время выполнения* операций.

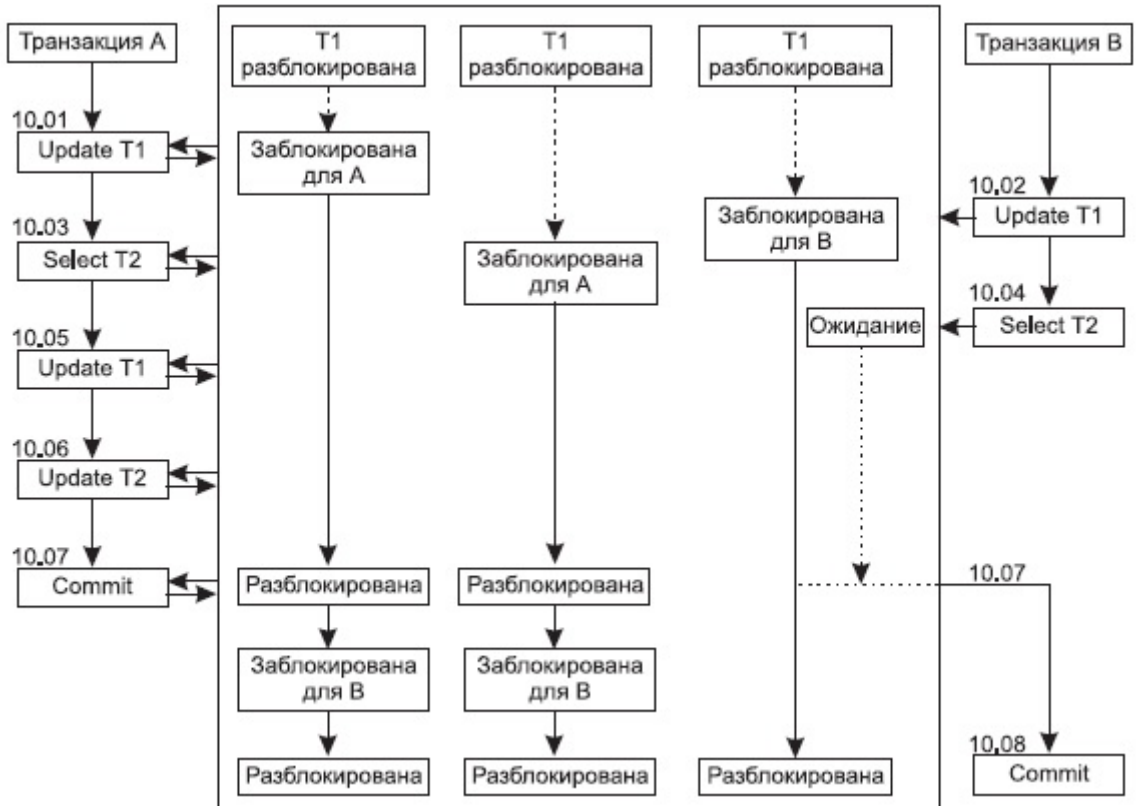


**Рис. 12.6.** Блокировки при одновременном выполнении двух транзакций

В общем случае на момент выполнения *транзакция* получает как бы монопольный *доступ* к объектам *БД*, с которыми она работает. В этом случае другие транзакции не получают доступа к объектам *БД* до момента окончания транзакции. Такой механизм действительно ликвидирует все перечисленные ранее проблемы: пропавшие изменения, неподтвержденные данные, несогласованные данные, строки-фантомы. Однако такая *блокировка* создает новые проблемы — задержку выполнения транзакций из-за блокировок.

Рассмотрим существующие типы конфликтов между двумя параллельными транзакциями. Можно выделить следующие типы:

- W-W — транзакция 2 пытается изменять объект, измененный не закончившейся транзакцией 1;
- R-W — транзакция 2 пытается изменять объект, прочитанный не закончившейся транзакцией 1;
- W-R — транзакция 2 пытается читать объект, измененный не закончившейся транзакцией 1.

Практические методы сериализации транзакций основываются на учете этих конфликтов.

Блокировки, называемые также синхронизационными захватами объектов, могут быть применены к разному типу объектов. Наибольшим объектом блокировки может быть вся *БД*, однако этот вид блокировки сделает *БД* недоступной для всех приложений, которые работают с данной *БД*. Следующий *тип объекта* блокировки — это таблицы. *Транзакция*, которая работает с таблицей, блокирует ее на все *время вы*полнения транзакции. Именно такой вид блокировки рассмотрен в примере 12.7. Этот вид блокировки предпочтительнее предыдущего, потому что позволяет параллельно выполнять транзакции, которые работают с другими таблицами.

В ряде *СУБД* реализована *блокировка* на уровне страниц. В этом случае *СУБД* блокирует только отдельные страницы на диске, когда *транзакция* обращается к ним. Этот вид блокировки еще более мягок и позволяет разным транзакциям работать даже с одной и той же таблицей, если они обращаются к разным страницам данных.

В некоторых *СУБД* возможна *блокировка на уровне строк*, однако такой механизм блокировки требует дополнительных затрат на поддержку этого вида блокировки.

В настоящее время проблема блокировок является предметом большого числа исследований.

Для повышения параллельности выполнения транзакций используется комбинирование разных типов синхронизационных захватов.

Рассматривают два типа блокировок (синхронизационных захватов):

- совместный режим блокировки — нежесткая, или *разделяемая, блокировка*, обозначаемая как **S** (Shared). Этот режим обозначает разделяемый захват объекта и требуется для выполнения операции чтения объекта. Объекты, заблокированные таким образом, не изменяются в ходе выполнения транзакции и доступны другим транзакциям также, но только в режиме чтения;

- монопольный режим блокировки — жесткая, или эксклюзивная, блокировка, обозначаемая как **X** (eXclusive). Данный режим блокировки предполагает монопольный захват объекта и требуется для выполнения операций занесения, удаления и модификации. Объекты, заблокированные данным типом блокировки, фактически остаются в монопольном режиме обработки и недоступны для других транзакций до момента окончания работы данной транзакции.

Захваты объектов несколькими транзакциями по чтению совместимы, то есть нескольким транзакциям допускается читать один и тот же *объект*, *захват* объекта одной транзакцией по чтению не совместим с захватом другой транзакцией того же объекта по записи, и захваты одного объекта разными

транзакциями по записи не совместимы. Правила совместимости захватов одного объекта разными транзакциями изображены на рис. 12.7:

| | | Транзакция B | | |
|---|---|---|---|---|
| | | Разблокирована | Нежесткая блокировка | Жесткая блокировка |
| Транзакция A | Разблокирована | Да | Да | Да |
| | Нежесткая блокировка | Да | Да | Нет |
| | Жесткая блокировка | Да | Нет | Нет |

**Рис. 12.7.** Правила применения жесткой и нежесткой блокировок транзакций

В примере, представленном на рис. 12.7 считается, что первой блокирует *объект транзакция* A, а потом пытается получить к нему *доступ транзакция* B.

На рис. 12.8 приведен ранее рассмотренный пример с выполнением транзакций 1 и 2, но с учетом разных типов блокировки. На рисунке видно, что, применив нежесткую блокировку к таблице 2 со стороны транзакции 1, мы обеспечили существенное уменьшение времени выполнения транзакции 2. Теперь *транзакция* 2 не ждет окончания транзакции 1, и поэтому завершает свою работу намного раньше.

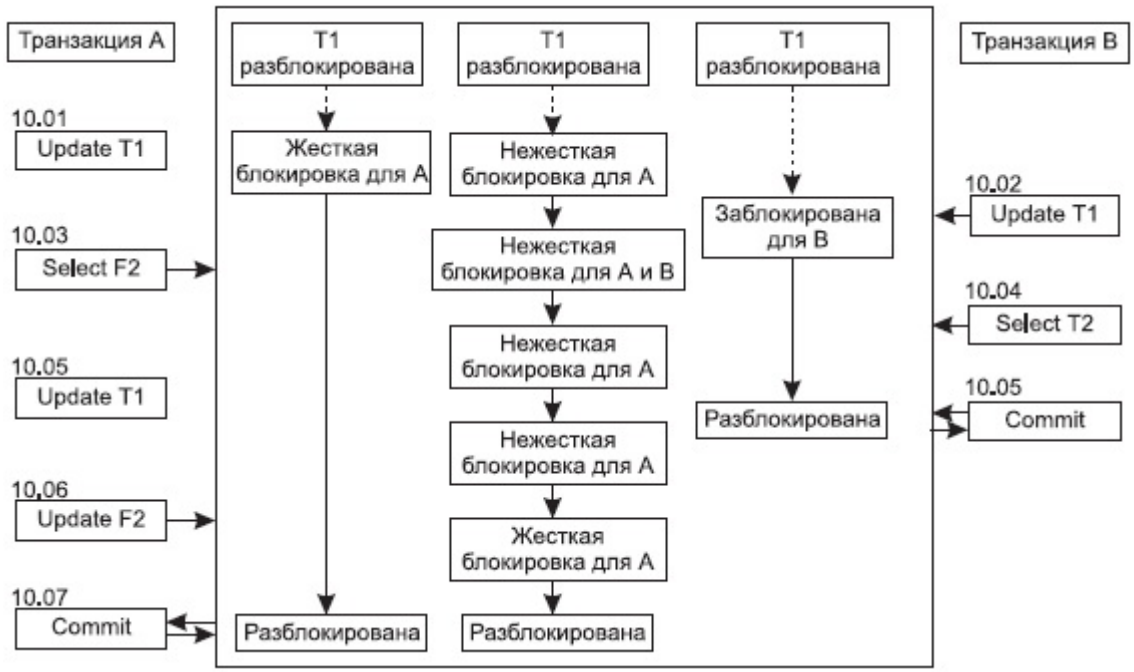


**Рис. 12.8.** Использование жесткой и нежесткой блокировки

К сожалению, применения разных типов блокировок приводит к проблеме тупиков. Эта проблема не нова. Проблема тупиков возникла при рассмотрении выполнения параллельных процессов в операционных средах и

также была связана с управлением разделяемыми (совместно используемыми) ресурсами.

Действительно, рассмотрим пример. Пусть *транзакция* A сначала жестко блокирует таблицу 1, а потом жестко блокирует таблицу 2. *Транзакция* B, наоборот, сначала жестко блокирует таблицу 2, а потом жестко блокирует таблицу 1. Если обе эти транзакции начали работу одновременно, то после выполнения операций модификации первыми объектами каждой транзакции они обе окажутся в бесконечном ожидании: *транзакция* A будет ждать завершения работы транзакции B и разблокировки таблицы 2, а *транзакция* B также безрезультатно будет ждать окончания работы транзакции A и разблокировки таблицы 1 (см. рис. 12.9).

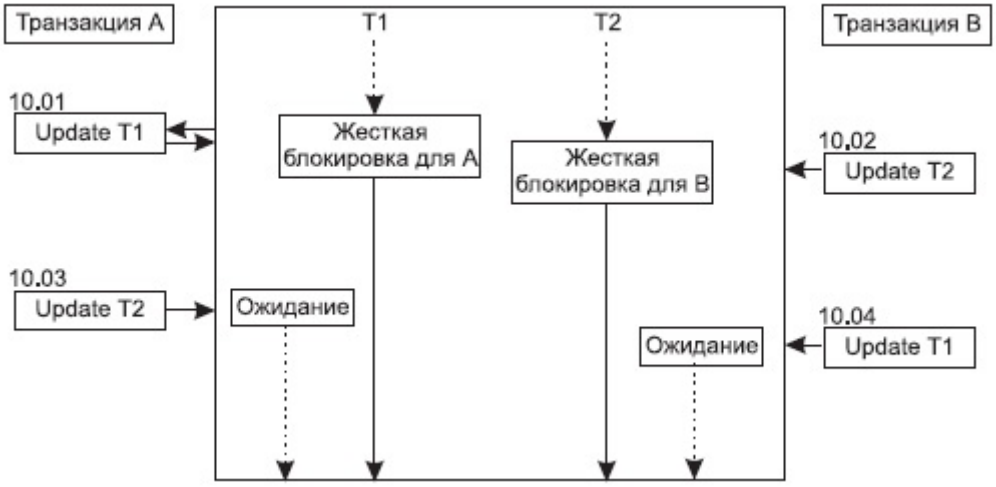


**Рис. 12.9.** Взаимная блокировка транзакций

Ситуации могут быть гораздо более сложными. Количество взаимно заблокированных транзакций может оказаться гораздо больше. Эту ситуацию каждая из транзакций обнаружить самостоятельно не может. Ее должна разрешить *СУБД*. И действительно, в большинстве коммерческих *СУБД* существует механизм обнаружения таких *тупиковых ситуаций*.

Основой обнаружения *тупиковых ситуаций* является построение (или постоянное поддержание) графа ожидания транзакций. *Граф* ожидания транзакций может строиться двумя способами. В книге К. Дж. Дейта *граф* ожидания — это *направленный граф*, в вершинах которого расположены имена транзакций. Если *транзакция* A ждет окончания транзакции B, то из вершины A в вершину B идет стрелка. Дополнительно стрелки могут быть помечены именами заблокированных объектов и типом блокировки. Пример такого графа ожиданий приведен на рис. 12.10.

Этот *граф* ожиданий построен для транзакций T1, T2,___,T12, которые работают с объектами *БД* A, B, ..., H.

Перечень действий, которые совершают транзакции над объектами, приведен в табл. 12.1.

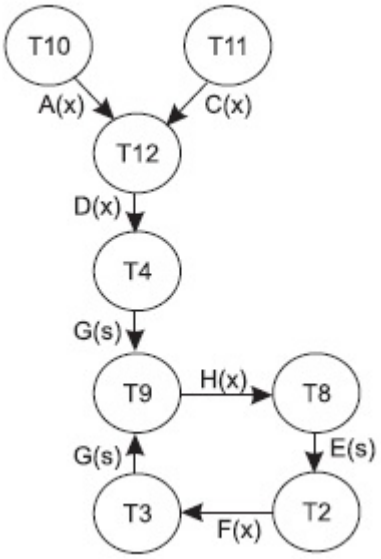


**Рис. 12.10.** Пример графа ожиданий транзакций

Таблица 12.1. Перечень действий множества транзакций

| **Время** | **Транзакция** | **Действие** |
|---|---|---|
| 0 | T1 | Select A |
| 1 | T2 | Select B |
| 2 | T1 | Select C |
| 3 | T4 | Select D |
| 4 | T5 | Select A |
| 5 | T2 | Select E |
| 6 | T2 | Update E |
| 7 | T3 | Select F |
| 8 | T2 | Select F |
| 9 | T5 | Update A |
| 10 | T1 | Commit |
| 11 | T6 | Select A |
| 12 | T5 | Commit |
| 13 | T6 | Select C |
| 14 | T6 | Update C |
| 15 | T7 | Select G |
| 16 | T8 | Select H |
| 17 | T9 | Select G |
| 18 | T9 | Update G |
| 19 | T8 | Select E |
| 20 | T7 | Commit |

| | | |
|---|---|---|
| 21 | T9 | Select H |
| 22 | T3 | Select G |
| 23 | T10 | Select A |
| 24 | T9 | Update H |
| 25 | T6 | Commit |
| 26 | T11 | Select C |
| 27 | T12 | Select D |
| 28 | T12 | Select C |
| 29 | T2 | Update F |
| 30 | T11 | Update C |
| 31 | T12 | Select A |
| 32 | T10 | Update A |
| 33 | T12 | Update D |
| 34 | T2 | Select G |
| 35 | - | - |

На графе объекты блокировки помечены типами блокировок, S — нежесткая (*разделяемая) блокировка*, X — жесткая (эксклюзивная) *блокировка*.

На диаграмме состояний ожидания видно, что транзакции T9, T8, T2 и T3 образуют цикл. Именно наличие *цикла* и является признаком возникновения тупиковой ситуации. Поэтому в момент 3 перечисленные транзакции будут заблокированы.

Разрушение тупика начинается с выбора в цикле транзакций так называемой транзакции-жертвы, то есть транзакции, которой решено пожертвовать, чтобы обеспечить возможность продолжения работы других транзакций.

Критерием выбора является *стоимость* транзакции; жертвой выбирается самая дешевая *транзакция*. *Стоимость* транзакции определяется на основе многофакторной оценки, в которую с разными весами входят *время выполнения*, число накопленных захватов, приоритет.

После выбора транзакции-жертвы выполняется *откат* этой транзакции, который может носить полный или частичный характер. При этом, естественно, освобождаются захваты и может быть продолжено выполнение других транзакций.

В лекциях профессора С. Д. Кузнецова приводится несколько иной *принцип построения* графа ожидания. В этом случае *граф* ожидания транзакций строится в виде ориентированного *двудольного графа*, в котором существует два типа вершин — вершины, соответствующие транзакциям, и вершины, соответствующие объектам захвата. В этом графе существует *дуга*, ведущая из вершины- транзакции к вершине-объекту, если для этой транзакции существует удовлетворенный *захват* объекта. В графе существует *дуга* из

вершины-объекта к вершине-транзакции, если *транзакция* ожидает удовлетворения захвата объекта.

Для распознавания тупика здесь, так же, как и в первом методе, производится построение графа ожидания транзакций и в этом графе ищутся циклы. Традиционной техникой (для которой существует множество разновидностей) нахождения *циклов в ориентированном графе* является *редукция графа*.

Не вдаваясь в детали, *редукция* состоит в том, что прежде всего из графа ожидания удаляются все дуги, исходящие из вершин-транзакций, в которые не входят дуги из вершин-объектов. (Это как бы соответствует той ситуации, что транзакции, не ожидающие удовлетворения захватов, успешно завершились и освободили захваты.) Для тех вершин-объектов, для которых не осталось входящих дуг, но существуют исходящие, ориентация исходящих дуг изменяется на противоположную (это моделирует удовлетворение захватов). После этого снова срабатывает первый шаг, и так до тех пор, пока на первом шаге не прекратится удаление дуг. Если в графе остались дуги, то они обязательно образуют цикл.

Естественно, такое насильственное устранение *тупиковых ситуаций* является нарушением принципа изолированности пользователей.

Заметим, что в централизованных системах *стоимость* построения графа ожидания сравнительно невелика, но она становится слишком большой в по-настоящему распределенных *СУБД*, в которых транзакции могут выполняться в разных узлах сети. Поэтому в таких системах обычно используются другие методы сериализации транзакций.

Для обеспечения сериализации транзакций синхронизационные захваты объектов, произведенные по инициативе транзакции, можно снимать только при ее завершении. Это требование порождает двухфазный протокол синхронизационных захватов — 2PL(two *phase* lock) или *2PC* (two *phase commit*). В соответствии с этим протоколом выполнение транзакции разбивается на две фазы:

- первая фаза транзакции — накопление захватов;
- вторая фаза (фиксация или откат) — освобождение захватов.

В языке *SQL* введен оператор явной блокировки таблицы, который позволяет точно задать тип блокировки для всей таблицы. *Синтаксис операции* блокировки имеет вид:

```
LOCK TABLE имя_таблицы IN {SHARED | EXCLUSIVE} MODE
```

Имеет смысл блокировать таблицу полностью, когда выполняется операция множественной модификации одной таблицы, то есть когда в ней изменяется большое количество строк. Эта операция иногда называется пакетным обновлением.

Конечно, у блокировки таблицы есть тот недостаток, что все остальные транзакции должны ждать окончания обновления таблицы. Но режим пакетного обновления одной таблицы работает достаточно быстро, и общая *производительность* выполнения *множества* транзакций может даже повыситься в этом случае.

## 12.10. Уровни изолированности пользователей

Достаточно легко убедиться, что при соблюдении двухфазного протокола синхронизационных захватов действительно обеспечивается полная сериализация транзакций. Однако иногда приложению, которое выполняет транзакцию, не столько важны точные данные, сколько скорость выполнения запросов. Например, в системах поддержки принятия решений по электронным торгам важно просто иметь представление об общей картине торгов, на основании которого принимается решение об повышении или снижении ставок и т. д. Для смягчения требований сериализации транзакций вводится понятие уровня изолированности пользователя.

Уровни изолированности пользователей связаны с проблемами, которые возникают при параллельном выполнении транзакций и которые были рассмотрены нами ранее.

Всего введено 4 уровня изолированности пользователей. Самый высокий уровень изолированности соответствует протоколу сериализации транзакций, это уровень *SERIALIZABLE*. Этот уровень обеспечивает полную изоляцию транзакций и полную корректную обработку параллельных транзакций.

Следующий уровень изолированности называется уровнем подтвержденного чтения — REPEATABLE READ. На этом уровне транзакция не имеет доступа к промежуточным или окончательным результатам других транзакций, поэтому такие проблемы, как пропавшие обновления, промежуточные или несогласованные данные, возникнуть не могут. Однако во время выполнения своей транзакции вы можете увидеть строку, добавленную в БД другой транзакцией. Поэтому один и тот же запрос, выполненный в течение одной транзакции, может дать разные результаты, то есть проблема строк-призраков остается. Однако если такая проблема критична, лучше ее разрешать алгоритмически, изменяя алгоритм обработки, исключая повторное выполнение запроса в одной транзакции.

Второй уровень изолированности связан с подтвержденным чтением, он называется *READ COMMITED*. На этом уровне изолированности транзакция не имеет доступа к промежуточным результатам других транзакций, поэтому проблемы пропавших обновлений и промежуточных данных возникнуть не могут. Однако окончательные данные, полученные в ходе выполнения других транзакций, могут быть доступны нашей транзакции. При этом уровне изолированности транзакция не может обновлять строку, уже обновленную другой транзакцией. При попытке выполнить подобное обновление транзакция будет

отменена автоматически, во избежание возникновения проблемы пропавшего обновления.

И наконец, самый низкий уровень изолированности называется уровнем неподтвержденного, или *грязного, чтения*. Он обозначается как READ UNCOMMITED. При этом уровне изолированности текущая транзакция видит промежуточные и несогласованные данные, и также ей доступны строки-призраки. Однако даже-при этом уровне изолированности СУБД предотвращает пропавшие обновления.

В стандарте *SQL2* существует оператор задания уровня изолированности выполнения транзакции. Он имеет следующий синтаксис:

SET TRANSACTION IZOLATION LEVEL

[{SERIALIZABLE | REPEATABLE READ | READ COMMITED | READ UNCOMMITED}]

[{READ WRITE | READ ONLY }]

Дополнительно в этом операторе может быть указано, операции какого типа выполняются в транзакции. По умолчанию предполагается уровень *SERIALIZABLE*. Если задан уровень READ UNCOMMITED, то допустимы только операции чтения в транзакции, поэтому в этом случае нельзя установить операции READ WRITE. В табл. 12.11 приведено соответствие уровней изолированности транзакций и проблем, возникающих при параллельном выполнении транзакций.

Таблица 12.12. Уровни изолированности транзакций и проблемы многопользовательской работы

| **Уровень изоляции** | **Появление пропавших обновлений** | **Появление промежуточных данных** | **Появление несогласованных данных** | **Появление строк-призраков** |
|---|---|---|---|---|
| *SERIALIZABLE* | СУБД предотвращает | СУБД предотвращает | СУБД предотвращает | СУБД предотвращает |
| REPEATABLE READ | СУБД предотвращает | СУБД предотвращает | СУБД предотвращает | Может произойти |
| READ COMMITED | СУБД предотвращает | СУБД предотвращает | Может произойти | Может произойти |
| READ UNCOMMITED | СУБД предотвращает | Может произойти | Может произойти | Может произойти |

В разных коммерческих СУБД могут быть реализованы не все уровни изолированности, это необходимо выяснить в технической документации.

## 12.11. Гранулированные синхронизационные захваты

Мы уже говорили, что объектами блокирования могут быть объекты разного уровня, начиная с целой БД и заканчивая кортежем.

Понятно, что чем крупнее объект синхронизационного захвата (неважно, какой природы этот объект — логический или физический), тем меньше синхронизационных захватов будет поддерживаться в системе, и при этом, соответственно, будут меньшие накладные расходы. Более того, если выбрать в качестве уровня объектов для захватов файл или отношение, то будет решена даже проблема фантомов (если это не ясно сразу, посмотрите еще раз на формулировку проблемы фантомов и определение двухфазного протокола захватов).

Но вся беда в том, что при использовании для захватов крупных объектов возрастает вероятность конфликтов транзакций и тем самым уменьшается допускаемая степень их параллельного выполнения. Фактически при укрупнении объекта синхронизационного захвата мы умышленно огрубляем ситуацию и видим конфликты в тех ситуациях, когда на самом деле конфликтов нет. Действительно, если транзакция T1 обрабатывает первую, пятую и двадцатую строку в таблице R1, но блокирует всю таблицу, то транзакция T2, которая обрабатывает шестую и восьмую строки той же таблицы не сможет получить к ним доступ, хотя на уровне строк никаких конфликтов нет.

В большинстве современных систем используются покортежные, то есть построковые синхронизационные захваты.

Однако нелепо было бы применять покортежную блокировку в случае выполнения, например, операции удаления всего отношения или удаления всех строк в отношении.

Подобные рассуждения привели к понятию *гранулированных* синхронизационных захватов и разработке соответствующего механизма.

При применении этого подхода синхронизационные захваты могут запрашиваться по отношению к объектам разного уровня: файлам, отношениям и кортежам. Требуемый *уровень объекта* определяется тем, какая операция выполняется (например, для выполнения операции уничтожения отношения объектом синхронизационного захвата должно быть все отношение, а для выполнения операции удаления кортежа — этот кортеж). Объект любого уровня может быть захвачен в режиме **S** (разделяемом) или **X** (монопольном). Вводится специальный протокол *гранулированных* захватов и определены новые типы захватов: перед захватом объекта в режиме **S** или **X** соответствующий объект более высокого уровня должен быть захвачен в режиме **IS, IX** или **SIX**.

**IS** (Intented for Shared lock, предваряющий разделяемую блокировку) по отношению к некоторому *составному объекту* **O** означает намерение захватить некоторый входящий в **O** объект в совместном режиме. Например, при намерении читать кортежи из отношения **R** это отношение должно быть захвачено в режиме **IS** (а до этого в таком же режиме должен быть захвачен файл).

**IX** (Intented for *eXclusive lock*, предваряющий жесткую блокировку) по отношению к некоторому *составному объекту* **O** означает намерение захватить некоторый входящий в **O** объект в монопольном режиме. Например, при

намерении удалять кортежи из отношения **R** это отношение должно быть захвачено в режиме **IX** (а до этого в таком же режиме должен быть захвачен файл).

**SIX** (Shared, Intented for *eXclusive lock*, *разделяемая блокировка* объекта, предваряющая дальнейшие жесткие блокировки его составляющих) по отношению к некоторому *составному объекту* О означает совместный захват всего этого объекта с намерением впоследствии захватывать какие-либо входящие в него объекты в монопольном режиме. Например, если выполняется длинная операция просмотра отношения с возможностью удаления некоторых просматриваемых кортежей, то экономичнее всего захватить это отношение в режиме **SIX** (а до этого захватить файл в режиме **IS** ).

Весьма трудно описать словами все возможные ситуации. Приведем полную таблицу совместимости захватов, анализируя которую можно выявить все случаи (см. табл. 12.2).

Таблица 12.2. Матрица совместимости блокировок.

| **L1\L2** | **X** | **S** | **IX** | **IS** | **SIX** |
|---|---|---|---|---|---|
| Нет блокировки | Да | Да | Да | Да | Да |
| **X** | Нет | Нет | Нет | Нет | Нет |
| **S** | Нет | Да | Нет | Да | Нет |
| **IX** | Нет | Нет | Да | Да | Нет |
| **IS** | Нет | Да | Да | Да | Да |
| **SIX** | Нет | Нет | Нет | Да | Нет |

Протокол *гранулированных* захватов требует соблюдения следующих правил:

1. Прежде чем транзакция установит **S** -блокировку на данный кортеж, она должна установить блокировку **IS** или другую, более сильную блокировку на отношение, в котором содержится данный кортеж.

2. Прежде чем транзакция установит **X** -блокировку на данный кортеж, она должна установить **IX** -блокировку или другую более сильную блокировку на отношение, в которое входит кортеж.

Блокировка **L1** называется более сильной по отношению к блокировке **L2** тогда и только тогда, когда для любой конфликтной ситуации (Нет — недопустимо) в столбце блокировки **L2** в некоторой строке матрицы совместимости блокировок (см. табл. 12.2) существует также конфликт в столбце блокировки **L1** в той же строке.

Диаграмма приоритетов блокировок приведена на рис. 12.12.

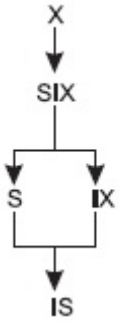


**Рис. 12.12.** Диаграмма приоритета блокировок различных типов

## 12.12. Предикатные синхронизационные захваты

Несмотря на привлекательность метода *гранулированных* синхронизационных захватов, следует отметить, что он не решает проблему фантомов (если, конечно, не ограничиться использованием захватов отношений в режимах **S** и **X** ).

Известно, что проблема фантомов не возникает, если объектом блокировки является целое отношение. Именно это свойство и послужило основой разработки метода предикатных синхронизационных захватов. В этом случае мы рассматриваем захват отношения — простой и частный случай предикатного захвата.

Суть этого метода — оценить множество кортежей, которое связано с той или иной транзакций, и если эти два множества, относящиеся к одному отношению, не пересекаются, то две транзакции могут оперировать ими параллельно без взаимной блокировки, а результаты выполнения обеих транзакций будут корректными.

Поскольку любая операция над реляционной базой данных задается некоторым условием (то есть в ней указывается не конкретный набор объектов базы данных, над которыми нужно выполнить операцию, а условие, которому должны удовлетворять объекты этого набора), идеальным выбором было бы требовать синхронизационный захват в режиме **S** или **X** именно этого условия. Но если посмотреть на общий вид условий, допускаемых, например, в языке SQL, то становится абсолютно непонятно, как определить совместимость двух предикатных захватов. Ясно, что без этого использовать предикатные захваты для синхронизации транзакций невозможно, а в общей форме проблема неразрешима.

К счастью, эта проблема сравнительно легко решается для случая простых условий. Будем называть простым условием конъюнкцию простых предикатов, имеющих вид:

имя-атрибута { операция сравнения } значение

Здесь операция сравнения: =, >, <

В типичных СУБД, поддерживающих двухуровневую организацию (языковой уровень и уровень управления внешней памяти), в интерфейсе подсистем управления памятью (которая обычно заведует и сериализацией транзакций) допускаются только простые условия. Подсистема языкового уровня производит компиляцию исходного оператора со сложным условием в последовательность обращений к ядру СУБД, в каждом из которых содержатся только простые условия. Следовательно, в случае типовой организации реляционной СУБД простые условия можно использовать как основу предикатных захватов.

Для простых условий совместимость предикатных захватов легко определяется на основе следующей геометрической интерпретации. Пусть R — отношение с атрибутами $a_1, a_2, ..., a_n$, а $m_1, m_2, ..., m_n$ — множества допустимых значений $a_1, a_2, ..., a_n$ соответственно (все эти множества — конечные). Тогда можно сопоставить R конечное n -мерное пространство возможных значений кортежей R. Любое простое условие "вырезает" m-мерный прямоугольник в этом пространстве ( $m <= n$ ).

Тогда **S-X, X-S, X-X** предикатные захваты от разных транзакций совместимы, если соответствующие прямоугольники не пересекаются.

Это иллюстрируется следующим примером, показывающим, что в каких бы режимах не требовала транзакция 1 захвата условия (1<=a<=4) & (b=5), а транзакция 2 — условия (1<=a<=5) & (1<=b<=3), эти захваты всегда совместимы.

Пример: (n = 2)

Заметим, что предикатные захваты простых условий описываются таблицами, немногим отличающимися от таблиц традиционных синхронизаторов.

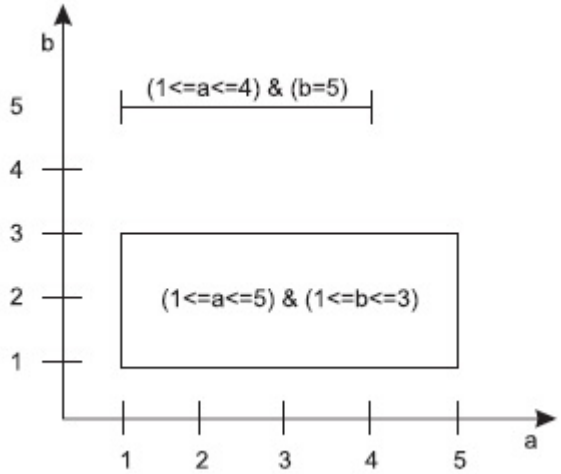


**Рис. 12.13.** Области действия предикатных захватов

## 12.13. Метод временных меток

Альтернативный метод сериализации транзакций, хорошо работающий в условиях редких конфликтов транзакций и не требующий построения графа ожидания транзакций, основан на использовании временных меток.

Основная идея метода (у которого существует множество разновидностей) состоит в следующем: если транзакция T1 началась раньше транзакции T2, то система обеспечивает такой режим выполнения, как если бы T1 была целиком выполнена до начала T2.

Для этого каждой транзакции T предписывается временная метка t, соответствующая времени начала T. При выполнении операции над объектом r транзакция T помечает его своей временной меткой и типом операции (чтение или изменение).

Перед выполнением операции над объектом r транзакция T1 выполняет следующие действия:

- Проверяет, не закончилась ли транзакция T, пометившая этот объект. Если T закончилась, T1 помечает объект r и выполняет свою операцию.
- Если транзакция T не завершилась, то T1 проверяет конфликтность операций. Если операции неконфликтны, при объекте r остается или проставляется временная метка с меньшим значением, и транзакция T1 выполняет свою операцию.
- Если операции T1 и T конфликтуют, то если t(T) > t(T1) (то есть транзакция T является более "молодой", чем T1 ), производится откат T и T1 продолжает работу.
- Если же t(T) < t(T1) ( T "старше" T1 ), то T1 получает новую временную метку и начинается заново.

К недостаткам метода временных меток относятся потенциально более частые откаты транзакций, чем в случае использования синхронизационных захватов. Это связано с тем, что конфликтность транзакций определяется более грубо. Кроме того, в распределенных системах не очень просто вырабатывать глобальные временные метки с отношением полного порядка (это отдельная большая наука).

Но в распределенных системах эти недостатки окупаются тем, что не нужно распознавать тупики, а как мы уже отмечали, построение графа ожидания в распределенных системах стоит очень дорого.