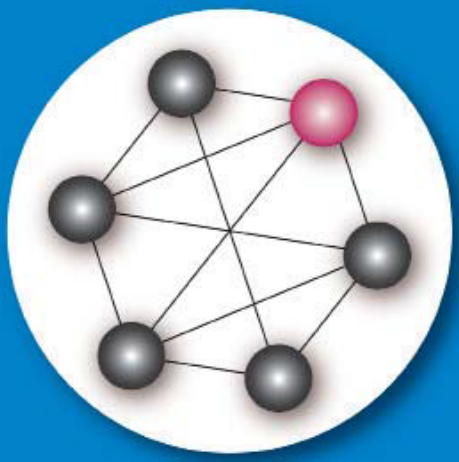


DBTechNet

DBTech VET

SQL Transactions

Теория и
практические упражнения

на русском языке

Lifelong Learning Programme

Данная публикация подготовлена в рамках проекта преподавателей
DBTech VET (DBTech VET). Код: 2012-1-FI1-LEO05-09365.

DBTech VET является проектом Leonardo da Vinci по многосторонней передаче инновационных проектов, финансируемым Европейской Комиссией и партнерами проекта Multilateral Transfer of Innovation.

www.DBTechNet.org
DBTech VET

Транзакции SQL – теория и практические упражнения
Первое издание, 2013 год, версия 1.3

Авторы: Мартти Лайхо, Димитрис А. Дервос, Кари Силпиё
Издание: проект преподавателей DBTech VET

ISBN 978-952-93-2420-0 (книга в мягкой обложке)
ISBN 978-952-93-2421-7 (электронная версия в формате PDF)

Транзакции SQL – учебное пособие для студента

Цели

Надежный доступ к данным должен быть основан на разработанных должным образом транзакциях SQL с учетом НУЛЕВОЙ ТОЛЕРАНТНОСТИ (абсолютной недопустимости) неверных данных в базе данных. Программист, не имеющий представления о необходимых технологиях транзакций, может легко нарушить целостность данных в базе данных, а также ухудшить или полностью свести к нулю производительность. По аналогии с правилами дорожного движения, необходимо знать и следовать правилам доступа к базам данных.

Целью этой обучающей программы является объяснение основ программирования транзакций с использованием основных продуктов СУБД (систем управления базами данных). Также в этом учебном пособии представлены характерные проблемы и возможности настройки транзакций.

Целевая группа

В целевую группу данного учебного пособия входят преподаватели, студенты профессионально-технических учебных заведений и промышленно-ориентированных высших учебных заведений. Данное учебное пособие также может быть полезно разработчикам приложений в информационно-коммуникационной индустрии для понимания СУБД, отличных от тех, которыми они пользуются каждый день.

Предпосылки к обучению

Читатели должны иметь практические навыки по использованию основ SQL некоторых продуктов СУБД.

Методы обучения

Учащимся предлагается поэкспериментировать и проверить себя с помощью заданий, представленных в этом пособии с использованием реальных СУБД. Для этих целей были составлены бесплатная виртуальная лаборатория базы данных и примеры скриптов, ссылки на которые предоставляются в разделе «Справки и ссылки».

Содержание

1 Транзакции SQL – рабочий логический блок

1.1 Введение в транзакции

В повседневной жизни люди осуществляют различные виды коммерческих сделок (транзакций), как например, покупка продуктов, заказ путешествий, изменение или отмена заказов, покупка билетов на концерты, оплата счетов за аренду, электричество, страховку и т. д. Разумеется, сделки (транзакци) имеют отношение не только к компьютерам. Любой вид человеческой деятельности, включающий в **себя рабочий логический блок**, который должен быть либо выполнен, либо быть отменен в целом, представляет собой **сделку, то есть транзакцию.**

Почти все информационные системы используют для хранения и извлечения данных услуги той или иной системы управления базами данных (СУБД). Современная СУБД - технически сложная система для обеспечения целостности данных, также она обеспечивает быстрый доступ к этим данным, в том числе и для нескольких одновременно работающих пользователей. Они обеспечивают надежное обслуживание приложений, управляющих хранением данных, но только если эти приложения используют надежные услуги надлежащим образом. Это осуществляется путем построения средств доступа к данным приложения, используя логику транзакции базы данных.

Неправильное управление транзакциями со стороны прикладного программного обеспечения может, например, послужить причиной:

- потери заказов, платежей клиентов, несостоявшейся отгрузки в интернет-магазине;
- сбоя в регистрации посадочных мест, или выполнение двойной регистрации пассажиров на поезд или самолет;
- невозможности звонка для вызова оперативных служб в центрах по чрезвычайным ситуациям и т.д.

Описанные выше проблемы нередко возникают в реальной жизни, но люди, несущие за это ответственность часто предпочитают не доводить эти случаи до сведения общественности. Целью проекта DBTech VET является создание основы передового опыта и методологии для предотвращения такого рода несчастных случаев.

С точки зрения манипуляции содержимым базы данных транзакции являются восстанавливаемыми блоками задач доступа к данным. Они также содержат элементы восстановления для всей базы данных в случае сбоя системы (см. приложение 3). Они также обеспечивают основу для управления согласованием в многопользовательской среде (параллельный или конкурирующий доступ).

1.2 Концепции «клиент-сервер» в среде SQL

В этом учебном пособии мы сосредоточимся на доступе к данным с помощью **транзакций SQL** при выполнении SQL-кода в интерактивном режиме, но не будем

упускать из виду, что соответствующий программный доступ к данным использует несколько иную парадигму (идеи и понятия, определяющие стиль написания компьютерных программ) программирования в отличие от той, что приведена в примерах в приложении 2.

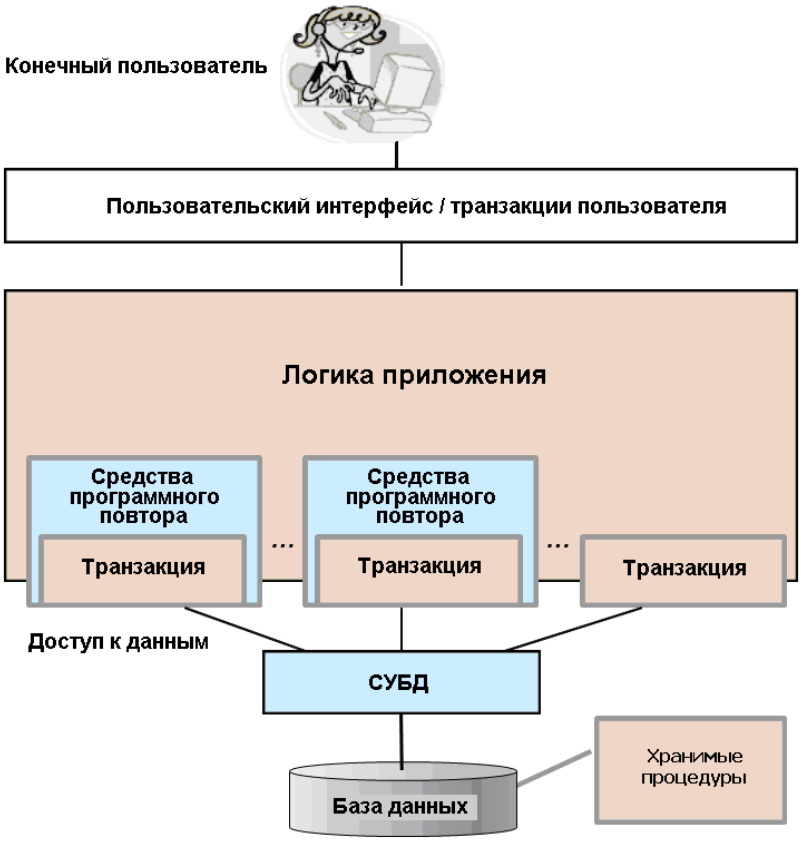


Рисунок 1.1 Расположение транзакций SQL в протоколах прикладного уровня

На рисунке 1.1 представлен упрощенный вид архитектуры типичного приложения баз данных, с расположением транзакций базы данных на слое программного обеспечения, отличном от слоя пользовательского интерфейса. С точки зрения конечного пользователя, один или более вариантов использования для обработки бизнес-операции могут быть определены для приложения и реализованы как **пользовательская транзакция**. Одна транзакция пользователя может включать в себя несколько транзакций SQL, некоторые из которых будут состоять из поиска и, как правило, окончательной транзакции в цепочке обновления содержимого базы данных. **Средства программного повтора** в логике приложения включают в себя возможности для реализации повторов действий в случае неудачи транзакций SQL в многопользовательской среде (параллельный или конкурирующий доступ).

Для правильного понимания транзакций SQL мы должны договориться о некоторых основных понятиях, касающихся квитирования (англ. «handshaking» - подтверждение приёма-передачи структурной единицы информации между клиентом и сервером). Для получения доступа к базе данных приложение должно инициировать соединение с ней, в свою очередь соединение устанавливает контекст SQL-сессии. Проще говоря, SQL-сессию инициирует **SQL-клиент**, а **SQL-сервер** содержит в себе сервер базы данных. С точки зрения сервера баз данных, приложение использует сервисы базы данных в режиме клиент-сервер, передавая **SQL команды** в качестве параметров функций или методов посредством доступа к данным API (англ. Application Programming Interface - интерфейс

программирования приложений)[1]. Независимо от того, какой интерфейс используется для доступа к данным, диалог "логического уровня" с сервером основан на языке SQL, а надежный доступ к данным реализуется надлежащим использованием SQL транзакций.

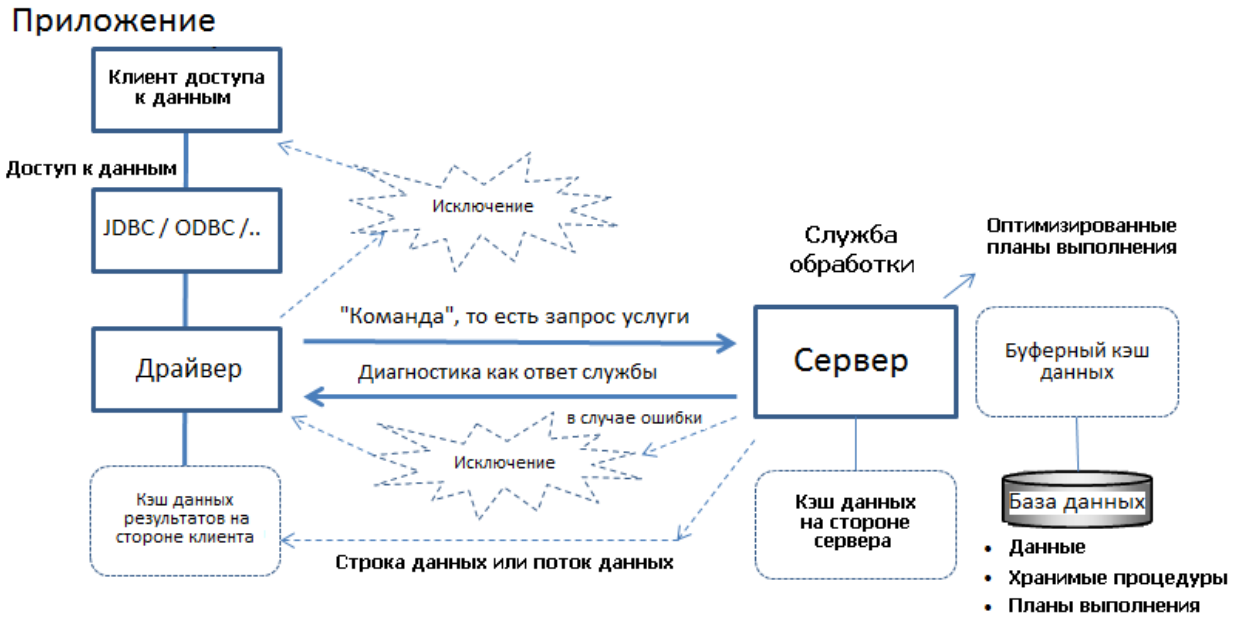


Рисунок 1.2 Пояснение обработки команд SQL

На рисунке 1.2 представлен полный цикл обработки команд SQL, начиная от создания клиентом **запроса к серверу** на обслуживание с использованием стека промежуточного программного обеспечения и сетевых сервисов, и заканчивая обработкой данных сервером и возвращаемым ответом на запрос. Команда SQL может включать одно или несколько **предложений SQL**. SQL- предложения обрабатываются, анализируются на основе метаданных базы данных, затем оптимизируются и, наконец, выполняются. Для компенсации ухудшения производительности по причине замедленности операций ввода/вывода диска, сервер сохраняет все недавно использованные строки в буферном пуле, находящемся в памяти RAM, где и происходит обработка всех данных.

Выполнение введенной команды SQL не может быть разделено или проведено на сервере частично, команда SQL должна быть выполнена полностью, в противном случае вся эта команда будет отменена (выполнен откат). В ответ на команду SQL, сервер посылает **диагностическое сообщение** (или сообщения) успешности или неудачи команды. Ошибки выполнения команды передаются клиенту в виде последовательности исключений. Однако, важно понимать, что такие SQL-операторы как UPDATE или DELETE будут успешно выполнены даже при отсутствии соответствующих строк. Следует отметить, что с точки зрения приложения такие случаи представляют собой неудачу, успешными же их можно считать ровно настолько, насколько выполнение команды доведено до конца. Поэтому код программы должен тщательно проверить диагностические сообщения, переданные сервером, чтобы определить число строк, которые были затронуты рассматриваемой операцией.

В случае применения оператора SELECT сгенерированный набор данных будет перенесён построчно на сторону клиента; его строки будут либо перенесены непосредственно от сервера по сети, или же они будут извлечены из кэша на стороне клиента.

[1] Например, ODBC, JDBC, ADO.NET, LINQ и другие, в зависимости от языка программирования, которым может выступать C++, C#, Java, PHP и т.д.

1.3 Транзакции SQL

Когда логика приложения должна выполнить последовательность команд SQL в неделимом виде, то команды должны быть сгруппированы в рабочий логический блок (англ. LUW - logical unit of work), называемую транзакция SQL, которая при обработке данных преобразует базу данных из одного согласованного состояния в другое и они, таким образом, могут рассматриваться как **элемент согласованности**. Любое успешное выполнение транзакции заканчивается командой **COMMIT** (фиксация), в то время как неудачное выполнение должно быть закончено командой **ROLLBACK** (откат), которая автоматически восстанавливает в базе данных все изменения, внесенные транзакцией. Таким образом, SQL транзакция может также рассматриваться в качестве **элемента восстановления**. Преимущество команды ROLLBACK (в стандартном SQL) состоит в том, что когда запрограммированная в транзакции логика приложения не может быть завершена, то нет никакой необходимости в проведении серии обратных операций отдельными командами, работа может быть просто отменена командой ROLLBACK, действие которой будет всегда успешно выполняться. Незавершенные транзакции в случае разрыва соединения, завершения программы или отказа системы будут автоматически выполнять откат системы. Также, в случае конфликтов в многопользовательской среде (параллельный или конкурирующий доступ), некоторые СУБД будут автоматически отзывать транзакцию, как это описано ниже.

Примечание: В соответствии со стандартом ISO SQL и как это реализовано, например, в DB2 и Oracle, любая команда SQL в начале SQL-сессии или после окончания транзакции, будет автоматически начинать новую транзакцию SQL. Этот случай представляет собой **неявное начало** транзакции SQL.

Некоторые продукты СУБД, например, SQL-сервер, MySQL/InnoDB, PostgreSQL и Pyrrho работают в режиме **AUTOCOMMIT** по умолчанию. Это означает, что результат каждой отдельной команды SQL будет автоматически фиксироваться в базе данных, таким образом эффекты и/или изменения, выполненные в базе данных рассматриваемым оператором, не могут быть отменены до прежнего состояния. Так, в случае ошибок приложение должно выполнить обратные операции для логической единицы работы, которые могут быть невозможными после операций параллельных (конкурирующих) SQL-клиентов. Также в случае прерванных соединений база данных может остаться в несогласованном состоянии. Для следования настоящей логике транзакций нужно каждую новую транзакцию начать с команды явного начала, например: BEGIN WORK, BEGIN TRANSACTION или START TRANSACTION, в зависимости от используемого продукта СУБД.

Примечание: В MySQL или InnoDB текущая сессия SQL может быть настроена на использование явных или неявных транзакций выполнением оператора:

SET AUTOCOMMIT = { 0 | 1 }

где 0 означает использование неявных транзакций, а 1 означает работу в режиме AUTOCOMMIT.

Примечание: Некоторые СУБД, такие как Oracle, фиксируют транзакцию неявным образом после выполнения любого оператора SQL DDL (например, CREATE, ALTER или DROP для некоторых объектов базы данных, таких как TABLE, INDEX, VIEW и т.д.).

Примечание: Весь SQL-сервер, включая его базы данных, может быть отконфигурирован для использования неявных транзакций и начатая сессия SQL (соединение) могут быть переключены на использование неявных транзакций, или же возвращены в режим AUTOCOMMIT выполнением оператора:

SET IMPLICIT_TRANSACTIONS { ON | OFF }

Примечание: Некоторые утилиты, такие как процессор командной строки (англ. command line processor - CLP) в IBM DB2 и некоторые интерфейсы доступа к данным, такие как ODBC и JDBC, работают по умолчанию в режиме AUTOCOMMIT. Например, в API JDBC, каждая транзакция должна быть начата следующим образом:

```
<connection>.setAutoCommit(false);
```

Вместо простой последовательности задач доступа к данным, некоторые транзакции SQL могут включать в себя набор программной логики. В таких случаях логика транзакции будет делать выбор во время выполнения, в зависимости от информации, полученной из базы данных. Даже в этом случае транзакция SQL может рассматриваться как неделимый рабочий логический блок (LUW), который либо успешно выполняется, либо отменяется. Тем не менее, сбой в транзакции обычно не генерирует автоматически команду ROLLBACK[2], но он должен быть диагностирован кодом приложения (см. далее «Диагностика ошибок SQL») и за необходимость применения команды ROLLBACK ответственно само приложение.

1.4 Логика транзакции

Давайте рассмотрим следующую таблицу банковских счетов:

```
CREATE TABLE Accounts (
acctId  INTEGER NOT NULL PRIMARY KEY,
balance DECIMAL(11,2) CHECK (balance >= 0.00)
);
```

Типичным примером учебника транзакций SQL является перевод определенной суммы (например, 100 евро) с одного счета на другой:

```
BEGIN TRANSACTION;
UPDATE Accounts SET balance = balance - 100 WHERE acctId = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctId = 202;
COMMIT;
```

В случае отказа системы или разрыва соединения «клиент-сервер» после первой команды UPDATE, протокол транзакции гарантирует, что деньги со счета номер 101 не будут потеряны, так как будет выполнен откат транзакции. Однако, этот пример транзакции далек от того, чтобы считаться надежным:

а) В случае, если бы один из этих двух банковских счетов не существовал, команды UPDATE были бы выполнены, что с точки зрения SQL является успешным

[2] Продукты СУБД в нашей лаборатории DebianDB после сбоя транзакции PostgreSQL будут принимать только команду ROLLBACK и отклонять все остальные команды.

завешением. Поэтому нужно изучить доступную диагностику SQL и проверить количество строк, которые были затронуты каждой из этих двух команд UPDATE.

b) В случае, если первая команда UPDATE потерпит неудачу из-за отрицательного баланса счета номер 101, (поскольку имело место нарушение соответствующего ограничения CHECK), то последующее продолжение и успешное выполнение второй команды UPDATE приведет к логически ошибочному состоянию в базе данных.

На этом простом примере мы видим и понимаем, что разработчики приложений должны знать о поведении СУБД, а также то, как изучается диагностика SQL доступа использователей к данным посредством интерфейса API. В любом случае здесь есть чему учиться и практиковаться в настройке транзакций.

1.5 Диагностика ошибок SQL

Вместо простой последовательности задач доступа к данным, некоторые транзакции SQL могут включать в себя набор программной логики. В таких случаях логика транзакции будет делать выбор во время выполнения, в зависимости от информации, полученной из базы данных. Даже в этом случае транзакция SQL может рассматриваться как неделимый рабочий логический блок (LUW), который либо успешно выполняется, либо отменяется. Тем не менее, сбой в транзакции обычно не генерирует автоматически команду ROLLBACK[3]. После каждой команды SQL код приложения должен проверить диагностические ошибки, выдаваемые сервером и определить, следует ли выполнить команду ROLLBACK или нет.

С этой целью ранний стандарт ISO SQL-89 определил команду SQLCODE в виде отображения целого числа, значение которого «0» в конце каждой команды SQL указывает на её успешное выполнение, в то время как значение 100 указывает на то, что соответствующие строки не были найдены, все другие значения могут отличаться от конкретного продукта СУБД. Любые положительные значения указывают на предупреждения, а отрицательные значения указывают на различные ошибки, на которые даны объяснения в соответствующих справочных руководствах продукта.

В стандарте ISO SQL-92 команда SQLCODE уже устарела и был введена новая команда SQLSTATE, строка из 5 символов, из которых первые два символа включают код класса SQL-ошибки и предупреждения, а последние 3 символа обозначают код подклассв. Наличие в строке пяти нулей ("00000") в SQLSTATE говорит об успешном выполнении. Были стандартизированы сотни других значений (например, нарушения ограничений SQL), но большое количество дополнительных значений принадлежит СУБД конкретного продукта. Значения SQLSTATE, начиная с «40» указывают на потерянные транзакции, например, из-за конфликта параллелизма, ошибки в записанной подпрограмме, прерванного соединения связи или проблемы с сервером.

Для обеспечения для клиентского приложения лучшей диагностической информацией касающейся того, что произошло на стороне сервера, X/Open Company Ltd расширила язык SQL оператором GET DIAGNOSTICS, который может использоваться для получения более детальной информации и может быть повторен для просмотра записей нескольких ошибок или предупреждений. Этот оператор был добавлен в ISO SQL стандарт SQL:1999, но только некоторая часть была реализована в таких СУБД, как например, DB2, Mimer и

[3] Продукты СУБД в нашей лаборатории DebianDB после сбоя транзакции PostgreSQL будут принимать только команду ROLLBACK и отклонять все остальные команды.

MySQL 5.6. Следующий пример из MySQL 5.6 показывает пример чтения элементов диагностики:

```
GET DIAGNOSTICS @rowcount = ROW_COUNT;
GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE,
                            @sqlcode  = MYSQL_ERRNO ;
SELECT @sqlstate, @sqlcode, @rowcount;
```

Некоторые реализации SQL с процедурными особенностями делают некоторые диагностические показатели доступными в специальных регистрах или функциях языка. Например, в MS SQL Server Transact-SQL (также называемый T-SQL), некоторые диагностические значения доступны в виде переменных @@, такие как @@ERROR для оригинального кода ошибки или @@ROWCOUNT для количества строк, которые были обработаны ранее.

В оригинале SQL языка IBM DB2, диагностические значения ISO SQL SQLCODE и SQLSTATE доступны в процедурных расширениях языка для хранимых процедур, как это показано ниже:

```
<SQL statement>
IF (SQLSTATE <> '00000') THEN
     <error handling>
END IF;
```

В блоках BEGIN-END языка PL/SQL, обработка ошибки (или исключения) кодируется в нижней секции кода посредством оператора EXCEPTION следующим образом:

```
BEGIN
     <processing>
EXCEPTION
WHEN <exception name> THEN
     <exception handling>;
...
WHEN OTHERS THEN
     err_code := sqlcode;
     err_text := sqlerrm;
     <exception handling>;
END;
```

Самая ранняя диагностическая запись имела отношение к реализациям, которые могут быть в ODBC, SQLExceptions и SQLWarnings, имеющих место в JDBC. В API JDBC языка Java, ошибки SQL повышают исключения SQL. То, что необходимо сделать, посредством управляющих структур «try-catch» в логике приложения выглядит следующим образом (см. приложение 2):

```
... throws SQLexception {
...
try {
      ...
      <JDBC statement(s)>
}
catch (SQLException ex) {
      <exception handling>
}
```

В JDBC диагностическое сообщение «rowcount», то есть число обработанных строк, возвращено выполнением методов.

1.6 Практические лабораторные задания

Примечание: Не верьте ничему, что вы прочитали! Для разработки надежных приложений вам необходимо **экспериментировать и сверяться** со службами вашего продукта СУБД. Различные СУБД отличаются способами, которыми они поддерживают даже самые основные службы транзакций SQL.

В приложении 1 мы приводим тесты явных и неявных транзакций SQL, команд COMMIT и ROLLBACK, а также логику транзакций с использованием MS SQL Server, но вам следует протестировать их самостоятельно, чтобы убедиться в поведении SQL Server, описанном в частях A1.1 - A1.2. Для проведения своих тестов вы можете скачать SQL Server Express бесплатно с веб-сайта Microsoft.

На первой лекции практических лабораторных занятий проходящий подготовку знакомится с использованием бесплатной виртуальной лаборатории DBTechNet базы данных в DebianDB, которая поставляется в комплекте с рядом предустановленных бесплатных СУБД, таких как IBM DB2 Express-C, Oracle XE, MySQL GA, PostgreSQL и Pyrrho. MySQL GA не является самым надежным продуктом СУБД в нашей лаборатории, но так как он широко используется в учебных заведениях, что мы в дальнейшем будем использовать его для проведения первого набора задач по обработке данных, основанных на транзакциях.

Итак, добро пожаловать в "Обзорный тур по тайнам транзакций» с использованием MySQL. Поведение MySQL зависит от того, какая именно СУБД используется. Ранние MySQL по умолчанию даже не поддерживали транзакции, но начиная с версии MySQL 5.1 ядром базы данных по умолчанию является InnoDB, которая поддерживает транзакции. Тем не менее, время от времени некоторые службы могут привести к неожиданным результатам.

Примечание: Серия экспериментов основывается на одной теме для всех СУБД в DebianDB, и они находятся в файле Appendix1_<dbms>.txt, который хранится в директории «**Transactions**» пользователя «student». Серия экспериментов не планируется как демонстрация проблем в MySQL, но мы не исключаем их появление, так как разработчики приложений также должны быть в курсе возникающих ошибок и уметь применять способы их устранения.

Мы предполагаем, что читатель уже ознакомился с документом DBTech VET «Введение в базы данных в Лаборатории DebianDB» (Краткое руководство), в котором приводится

описание процедуры установки и начальной настройки, которые требуются в DebianDB после установки на виртуальной машине (как правило, это Oracle VirtualBox).

После запуска DebianDB пользователь уже находится в системе с именем пользователя по умолчанию, (имя пользователя = **student**, пароль = **password**). Для того, чтобы создать первую базу данных MySQL, пользователю необходимо переключить пользователя (имя пользователя = **root**, пароль = **P4ssw0rd**), как это описано в кратком руководстве пользователя. Это можно сделать, кликнув на иконку «Terminal/Use the command line» («Терминал/Использование командной строки»), в верхней панели меню «Virtual Machine» («Виртуальная машина») (рис. 1.4).

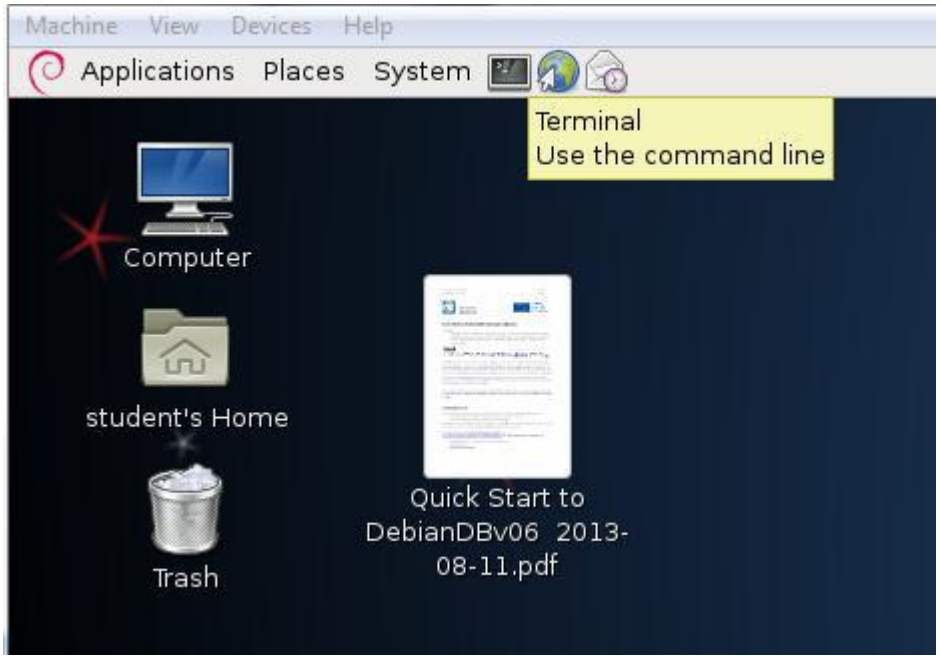


Рисунок 1.4 Иконка меню виртуальной машины «Terminal /Use the command line»

Затем, для запуска программы клиента **mysql** в окне терминала / командной строки необходимо ввести следующие команды Linux (рисунок 1.5):

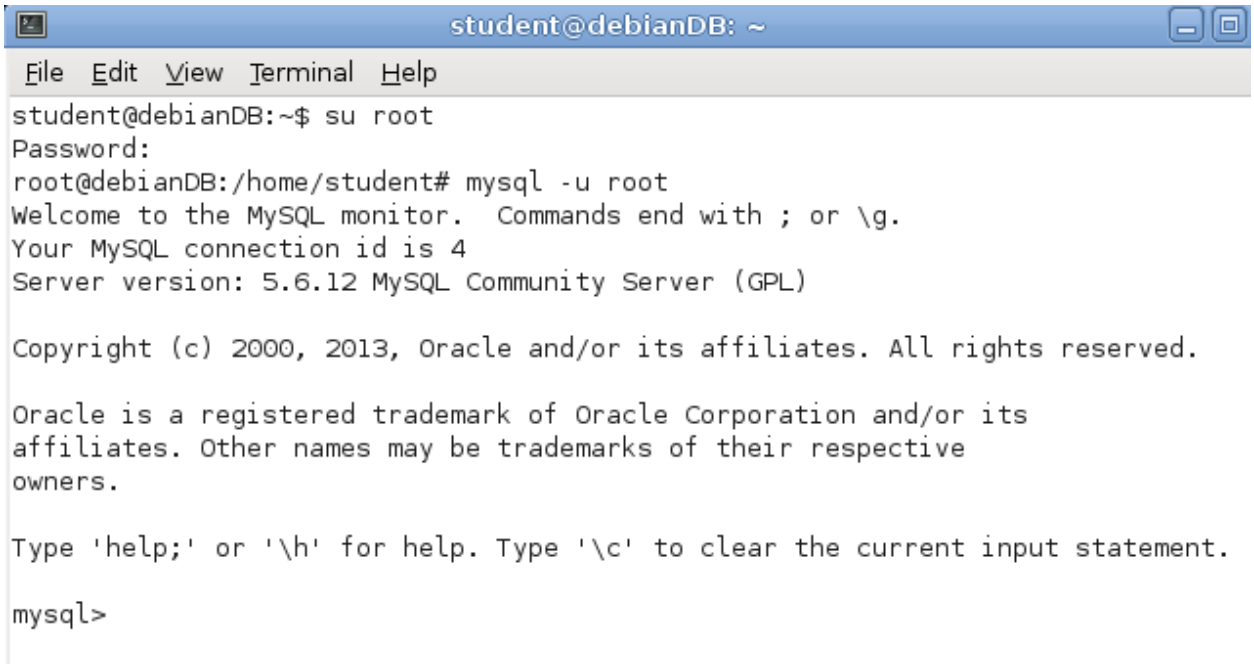


Рисунок 1.5 Начало сессии MySQL пользователем «**root**»

Следующая команда SQL, которую следует ввести для создания новой базы данных с именем «TestDB», выглядит следующим образом:

```
-----------------------------------
CREATE DATABASE TestDB;
-----------------------------------
```

Чтобы предоставить доступ и все возможные привилегии в созданной базе данных «TestDB», пользователь «root» должен ввести следующую команду:

```
-----------------------------------
GRANT ALL ON TestDB.* TO 'student'@'localhost';
-----------------------------------
```

Теперь необходимо ввести следующие две команды, чтобы выйти из пользователя «root» и завершить сессию MySQL, с последующим возвратом в сессию пользователя «student»:

```
-----------------------------------
EXIT;
exit
-----------------------------------
```

Примечание: Если во время сессии практических занятий экран DebianDB становится черным, на котором появляется форма для ввода пароля, в эту форму необходимо ввести пароль пользователя «student», которым является «password».

Теперь пользователь по умолчанию «student» может запустить клиент MySQL и получить доступ к базе данных TestDB следующим образом:

```
-----------------------------------
mysql
use TestDB;
-----------------------------------
```

Это будет началом новой сессии MySQL.

УПРАЖНЕНИЕ 1.1

Теперь мы составим новую таблицу с именем «T», имеющей три столбца: id (целое число, первичный ключ), s (строка с буквенными символами имеющая длину от 1 до 40 знаков), и si (в виде малого целого числа):

```
-----------------------------------
mysql> CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(40), si
SMALLINT);

Query OK, 0 rows affected (0.27 sec)
-----------------------------------
```

После каждой команды SQL клиент MySQL отображает определённую диагностику выполнения.

Чтобы убедиться в существовании таблицы, имеющую желаемую структуру, пригодится команда «DESCRIBE»:

```
-----------------------------------
DESCRIBE T;
-----------------------------------
```

> **Примечание**: MySQL в Linux чувствительна к регистру, за исключением таблиц и имен баз данных. Это означает, что будут работать следующие команды: «Describe T», «describe T», «create TaBle T ...», но «use testDB» и «describe t» относятся к базе данных и таблице, отличных от тех, которые упомянуты в этой практической работе.

Настало время, чтобы добавить / вставить несколько строк в только что созданную таблицу:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (1, 'first');
INSERT INTO T (id, s) VALUES (2, 'second');
INSERT INTO T (id, s) VALUES (3, 'third');
SELECT * FROM T ;
-----------------------------------
```

Команда «SELECT * FROM T» подтверждает, что три новых строки были добавлены в таблицу (обратите внимание на значения NULL, зарегистрированные в колонке "si").

> **Примечание**: Перед нажатием «enter» всегда удостоверяйтесь, что в конце каждой команды напечатали точку с запятой «;».

Имейте в виду, что для отмены текущей транзакции необходимо выполнить следующую команду:

```
-----------------------------------
ROLLBACK;
SELECT * FROM T ;
-----------------------------------
```

На первый взгляд это работает, но после ввода новой команды «SELECT * FROM T» мы видим, что таблица продолжает регистрацию трех строк. К нашему великому удивлению...

У причины нашего удивления есть имя – «AUTOCOMMIT». MySQL запускается в режиме AUTOCOMMIT, в котором каждая транзакция должна быть начата командой «START TRANSACTION», а после завершения транзакции MySQL снова возвращается в режим AUTOCOMMIT. Чтобы проверить это, следует выполнить следующий набор команд SQL:

```
-----------------------------------
START TRANSACTION;
INSERT INTO T (id, s) VALUES (4, 'fourth');
SELECT * FROM T ;
ROLLBACK;

SELECT * FROM T;
-----------------------------------
```

Вопрос

- Сравните результаты, полученные путем выполнения указанных выше двух команд «SELECT * FROM T».

УПРАЖНЕНИЕ 1.2

Далее мы выполним следующие команды:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (5, 'fifth');
ROLLBACK;
SELECT * FROM T;
-----------------------------------
```

Вопросы

- Каков результат, полученный после выполнения вышеприведенной команды «SELECT * FROM T»?
- Какой вывод (какие выводы) можно сделать про наличие возможных ограничений в использовании команды «START TRANSACTION» в MySQL / InnoDB?

УПРАЖНЕНИЕ 1.3

Теперь отключим режим автоматической фиксации AUTOCOMMIT, используя команду «SET AUTOCOMMIT»:

```
-----------------------------------
SET AUTOCOMMIT = 0;
-----------------------------------
```

Во-первых, удалим все строки таблицы, кроме одной:

```
-----------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;
-----------------------------------
```

Теперь снова вставим новые строки:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (6, 'sixth');
INSERT INTO T (id, s) VALUES (7, 'seventh');
SELECT * FROM T;
-----------------------------------
```

… а теперь отмена:

```
-----------------------------------
ROLLBACK;
SELECT * FROM T;
-----------------------------------
```

Вопрос

- В чем преимущество / недостаток использования команды «SET TRANSACTION» по сравнению с командой «SET AUTOCOMMIT», используемой для отключения установленного по умолчанию режима AUTOCOMMIT в MySQL?

Примечание: Когда клиент MySQL находится в окне терминала Linux, можно использовать клавиши клавиатуры «вниз» и «вверх», чтобы перелистывать вперед и назад ранее введённые команды. Следует отметить, что это не всегда возможно в другимх средах СУБД, предустановленными в DebianDB.

Примечание: Два последовательных символа тире « - » в SQL представляет собой сигнал, что остальная часть командной строки является комментарием, то есть текст, следующий за двумя тире, вплоть до следующего нажатия (символа) «enter» анализатором MySQL будет игнорироваться.

УПРАЖНЕНИЕ 1.4

```
-- Initializing only in case you want to repeat the exercise 1.4
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
DROP TABLE T2; --
COMMIT;
```

Строка комментария:

1. *Initializing only in case you want to repeat the exercise 1.4 - Инициализация только в случае, если вы хотите повторить упражнение 1.4*

Уже известно, что некоторые команды SQL относятся к категории Data Definition Language (DDL, англ. «язык описания данных», некоторые другие относятся к категории Data Manipulation Language (DML, англ. язык управления (манипулирования) данными). Примерами команд DDL являются команды CREATE TABLE, CREATE INDEX и DROP TABLE, а примерами DML являются команды SELECT, INSERT и DELETE. Зная об этом, далее стоит исследовать «глубину охвата» команды ROLLBACK, которая заключается в следующем:

```
SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (9, 'will this be committed?');
CREATE TABLE T2 (id INT);
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;

SELECT * FROM T; -- What has happened to T?
SELECT * FROM T2; -- What has happened to T2?
-- Compare this with SELECT from a missing table as follows:
SELECT * FROM T3; -- assuming that we have not created table T3

SHOW TABLES;
DROP TABLE T2;
COMMIT;
```

Строки комментариев:

1. *What has happened to T? - Что произошло с T?*
2. *What has happened to T2? - Что произошло с T2?*
3. *Compare this with SELECT from a missing table as follows: - Сравните с отсутствующей таблицей посредством команды SELECT следующим образом:*
4. *assuming that we have not created table T3 - если предположить, что мы еще не создали таблицу T3*

Вопрос

- Какой вывод (какие выводы) можно сделать?

УПРАЖНЕНИЕ 1.5

Содержимое таблицы T возвращается в исходное состояние:

```
-----------------------------------
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
COMMIT;
SELECT * FROM T;
COMMIT;
-----------------------------------
```

Теперь настало время проверить, вызывает ли возникновение ошибки автоматический откат (ROLLBACK) в MySQL. Для этого следует выполнить следующие команды SQL:

```
-----------------------------------
SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (2, 'Error test starts here');
-- division by zero should fail
SELECT (1/0) AS dummy FROM DUAL;
-- Now update a non-existing row
UPDATE T SET s = 'foo' WHERE id = 9999 ;
-- and delete an non-existing row
DELETE FROM T WHERE id = 7777 ;
--
INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
INSERT INTO T (id, s)
       VALUES (3, 'How about inserting too long of a string value?');
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
SHOW ERRORS;
SHOW WARNINGS;
INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
SELECT * FROM T;
COMMIT;

DELETE FROM T WHERE id > 1;
SELECT * FROM T;
COMMIT;
-----------------------------------
```

Строки комментариев:

1. *division by zero should fail - деление на нуль должно потерпеть неудачу*

2. *Now update a non-existing row - теперь обновление несуществующего ряда*
3. *and delete an non-existing row - и удаление несуществующего ряда*

Вопросы

a) Что мы выяснили про автоматический откат ошибки в MySQL?
b) Является ли деление на ноль ошибкой?
c) Реагирует ли MySQL на переполнения (overflow)?
d) Что мы узнаем из следующих результатов:

```
-----------------------------------
mysql> UPDATE T SET s = 'foo' WHERE id = 9999 ;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0 Changed: 0 Warnings: 0
mysql> INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
ERROR 1062 (23000): Duplicate entry '2'
-----------------------------------
```

В этом случае значение «23000», отображаемое клиентом MySQL, является нормированным значением SQLSTATE, указывающим на нарушения ограничения первичного ключа, а «1062» является соответствующим кодом ошибки продукта MySQL.

Как видно из нашего предыдущего примера, диагностика ошибки команды INSERT может быть вызвана с помощью новой команды «GET DIAGNOSTICS», доступной в MySQL начиная с версии 5.6, как это показано ниже:

```
mysql> GET DIAGNOSTICS @rowcount = ROW_COUNT;
Query OK, 0 rows affected (0.00 sec)

mysql> GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE,
    ->                             @sqlcode = MYSQL_ERRNO ;
Query OK, 0 rows affected (0.00 sec)
```

Переменные, начинающиеся с символа «@», являются нетипизированными локальными переменными в языке SQL. Мы используем локальные переменные MySQL в наших упражнениях только для того, чтобы моделировать уровень приложения, поскольку в этой книге мы работаем, главным образом, с языковым уровнем SQL. В доступе к данным интерфейса программирования приложений значения диагностики могут быть считаны непосредственно в переменных хоста, но следующий пример показывает, что мы можем считать значения и в локальных переменных:

```
mysql> SELECT @sqlstate, @sqlcode, @rowcount;
+-----------+----------+-----------+
| @sqlstate | @sqlcode | @rowcount |
+-----------+----------+-----------+
| 23000     |     1062 |        -1 |
+-----------+----------+-----------+
1 row in set (0.00 sec)
```

УПРАЖНЕНИЕ 1.6

```
DROP TABLE Accounts;
SET AUTOCOMMIT=0;
```

Текущая версия MySQL не поддерживает синтаксис ограничений CHECK на уровне столбцов, которую мы использовали для других СУБД для этого упражнения, а именно:

```
CREATE TABLE Accounts (
acctID INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL CONSTRAINT unloanable_account CHECK (balance >= 0)
);
```

Синтаксис **CHECK уровня строки** принимается следующим образом:

```
CREATE TABLE Accounts (
acctID INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL ,
CONSTRAINT unloanable_account CHECK (balance >= 0)
) ENGINE = InnoDB;
```

Но даже если синтаксис принимается, он не использует ограничения CHECK, как это видно из следующего эксперимента, который потерпит неудачу:

```
INSERT INTO Accounts (acctID, balance) VALUES (100,-1000);
SELECT * FROM Accounts;
ROLLBACK;
```

Примечание: Мы не удалили ограничение CHECK, чтобы сохранять возможность сопоставления экспериментов с другими продуктами,. У всех продуктов есть определённые ошибки, которые разработчики приложений должны устранить. Проблема несуществующей поддержки CHECK может быть решена созданием триггеров базы данных SQL, изучение которых выходит за рамки данного учебного материала, но заинтересованные читатели найдут наши примеры работ в файле скрипта AdvTopics_MySQL.txt.

```
-----------------------------------
-- Let’s load also contents for our test:
SET AUTOCOMMIT=0;
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
SELECT * FROM Accounts;
COMMIT;

-- A. Let's try the bank transfer
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;
SELECT * FROM Accounts;
ROLLBACK;

-- B. Let's test that the CHECK constraint actually works:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
SELECT * FROM Accounts ;
ROLLBACK;
```

Следующая транзакция SQL предназначена для проверки перевода суммы в 500 евро с банковского счета номер 101 на несуществующий номер банковского счета, имеющий, например, acctID = 777:

```
-- C. Updating a non-existent bank account 777:
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;
SELECT * FROM Accounts ;
ROLLBACK;
-----------------------------------
```

Строки комментариев:

1. *Let's load also contents for our test - загрузим содержание для нашего теста*
2. *Let's try the bank transfer - попробуем осуществить банковский перевод*
3. *Let's test that the CHECK constraint actually works – проверим действительно ли ограничение CHECK работает на самом деле*
4. *Updating a non-existent bank account 777 - обновление несуществующего банковского счета номер 777*

Вопросы

a) Выполняются ли две команды UPDATE, несмотря на то, что вторая соответствует запросу обновления несуществующего счета / строки в таблице счетов?

b) Если бы команда ROLLBACK в примере транзакции B или C была заменена на COMMIT, стала бы соответствующая транзакция успешной, сделав своё воздействие на базу данных постоянным?

c) Какие диагностические показатели MySQL пользовательское приложение может использовать для обнаружения проблем в приведённых выше транзакциях?

УПРАЖНЕНИЕ 1.7 – Транзакция SQL как элемент восстанавления

Теперь мы поэкспериментируем со свойствами «элемента восстановления» транзакций SQL в случае незавершенных транзакций. Чтобы осуществить это, мы начнем транзакцию, а затем разорвём соединение клиента с MySQL, после чего снова подключимся к базе данных, чтобы увидеть, какие изменения были внесены в базу незавершёнными транзакциями.

Обратите внимание на то, как на активные (то есть. незавершенные) транзакции влияет разрыв соединения (которые нередки случае с интернет-приложениями):

Во-первых, вставляется новая строка в T:

```
-----------------------------------
SET AUTOCOMMIT = 0;
INSERT INTO T (id, s) VALUES (9, 'Let''s see what happens if ..');
SELECT * FROM T;
-----------------------------------
```

Затем сессия клиента принудительно прерывается посредством команды «Control-C» (Ctrl-C) (рисунок 1.6):

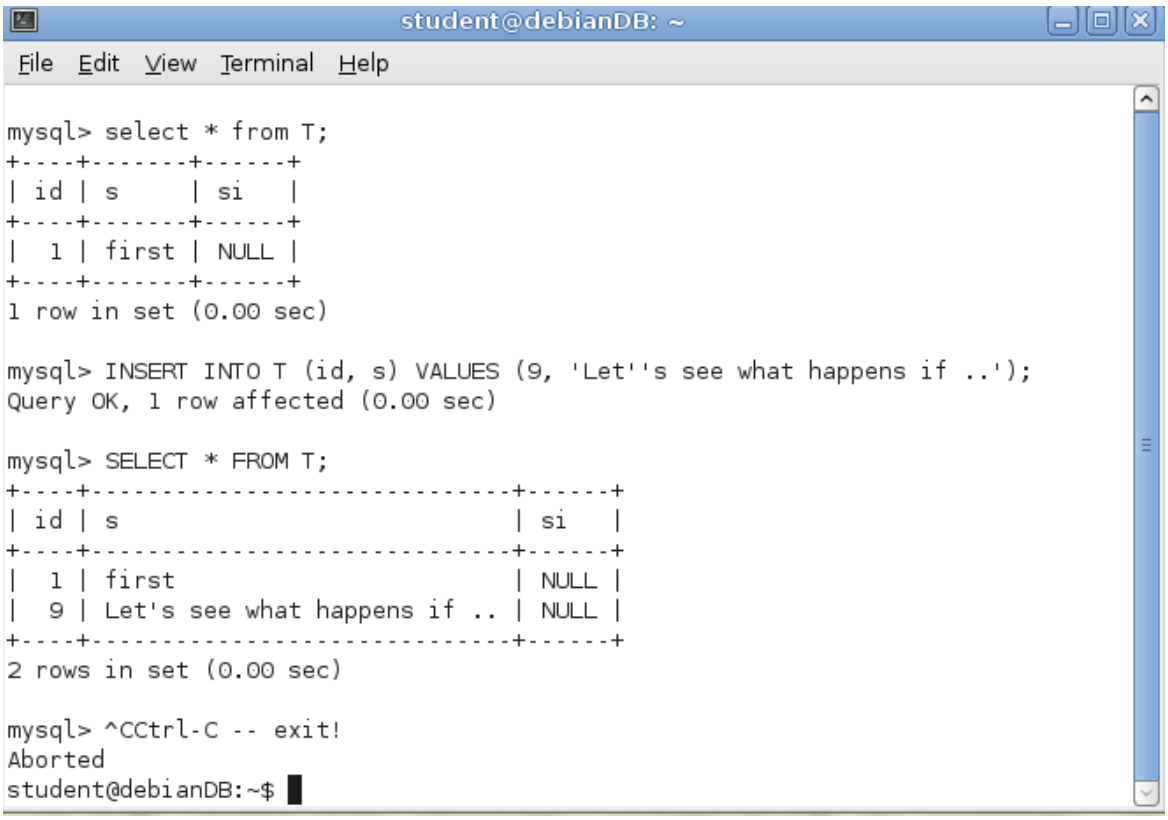


Рисунок 1.6 Симуляция отказа СУБД

Затем окно терминала DebianDB закрывается и открывается новое, начата новая сессия MySQL с базой данных TestDB:

```
-----------------------------------
mysql
USE TestDB;
SET AUTOCOMMIT = 0;
SELECT * FROM T;
COMMIT;
EXIT;
-----------------------------------
```

Вопрос

- Какие будут комментарии о содержании строки в T на этот раз?

Примечание: Все изменения, внесенные в базу данных прослеживаются в **журнале транзакций** базы данных. Приложение 3 объясняет, как серверы баз данных используют журналы транзакций для восстановления базы данных до последней зафиксированной транзакции, предшествовавшей сбою системы. Упражнение 1.7 может быть продолжено для проверки сбоя системы, если вместо отмены одного только клиента MySQL будет прерван процесс MySQL сервера **mysqld**:

```
student@debianDB:~$ su root
Password:
root@debianDB:/home/student# ps -e | grep mysqld
1092 ? 00:00:00 mysqld_debianDB
1095 ? 00:00:00 mysqld_safe
1465 ? 00:00:01 mysqld
root@debianDB:/home/student# kill 1465
```

2 Конкурирующие транзакции

Небольшой совет
Не верьте ничему, что вы слышали или читали о поддержке транзакций в различных СУБД! В целях разработки надежных приложений, необходимо экспериментировать и самостоятельно проверять функциональность сервисов, поддерживаемых СУБД. СУБД различаются по способу их реализации и поддержки даже базовых сервисов транзакций SQL.

Прикладная программа, корректно работающая в однопользовательской окружающей среде, может неожиданно столкнуться с проблемами надёжности работы в многопользовательской окружающей среде, одновременно обслуживая несколько клиентов, как это представлено на рисунке 2.1.

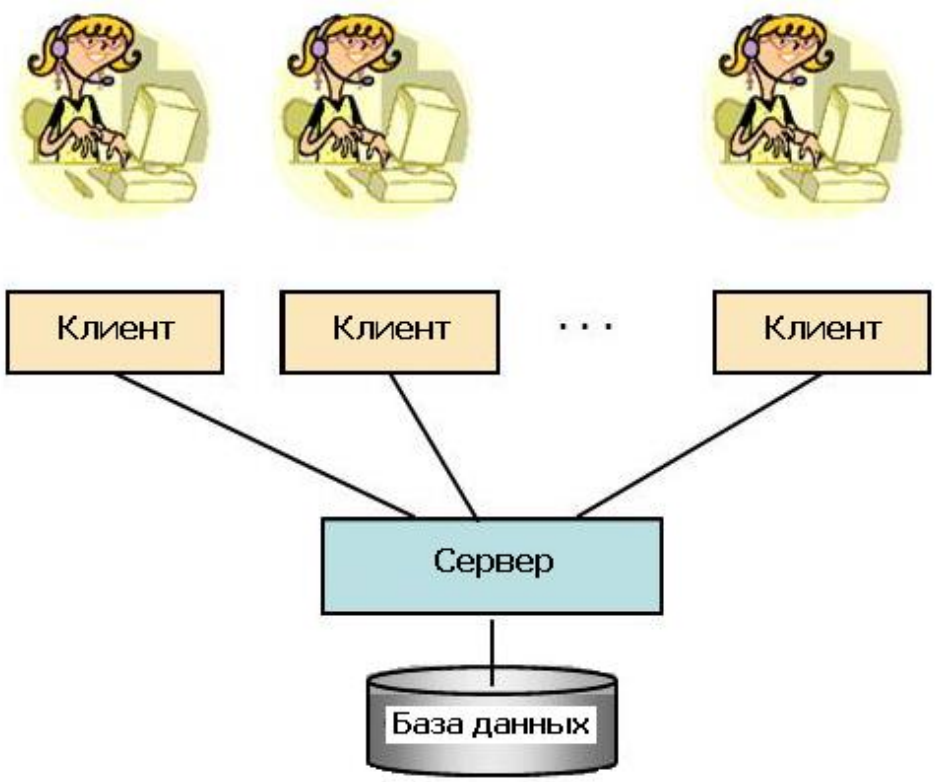


Рисунок 2.1 Доступ к одной базе данных нескольких клиентов в многопользовательской среде

2.1 Проблемы параллелизма - возможные риски надежности

Не имея надлежащих служб управления параллелизмом в продукте базы данных или при недостатке в знании о том, как использовать услуги должным образом, **содержимое** базы данных или **результаты** наших запросов могут быть повреждены, то есть стать **ненадежными**.

Далее мы рассмотрим типичные проблемы (аномалии) параллелизма:

- Проблема потерянного обновления;
- Проблема считывания «грязных» данных, то есть чтение незафиксированных данных некоторых параллельных транзакций;

- Проблема неповторяющегося чтения, т. е. повторяющееся считывание не возвращает те же строки и/или их значения;
- Проблема фантомного чтения, то есть во время транзакции некоторые выбранные строки могут быть не видны транзакцией.

После чего мы представим решения этих проблем в соответствии со стандартом ISO SQL и реальных продуктов СУБД.

2.1.1 Проблема потерянного обновления

Си Джи Дэйт представил следующий пример на рисунке 2.2, где два пользователя в различных банковских автоматах (ATM) снимают деньги с одного и того же банковского счета, изначальный баланс которого 1000 евро.

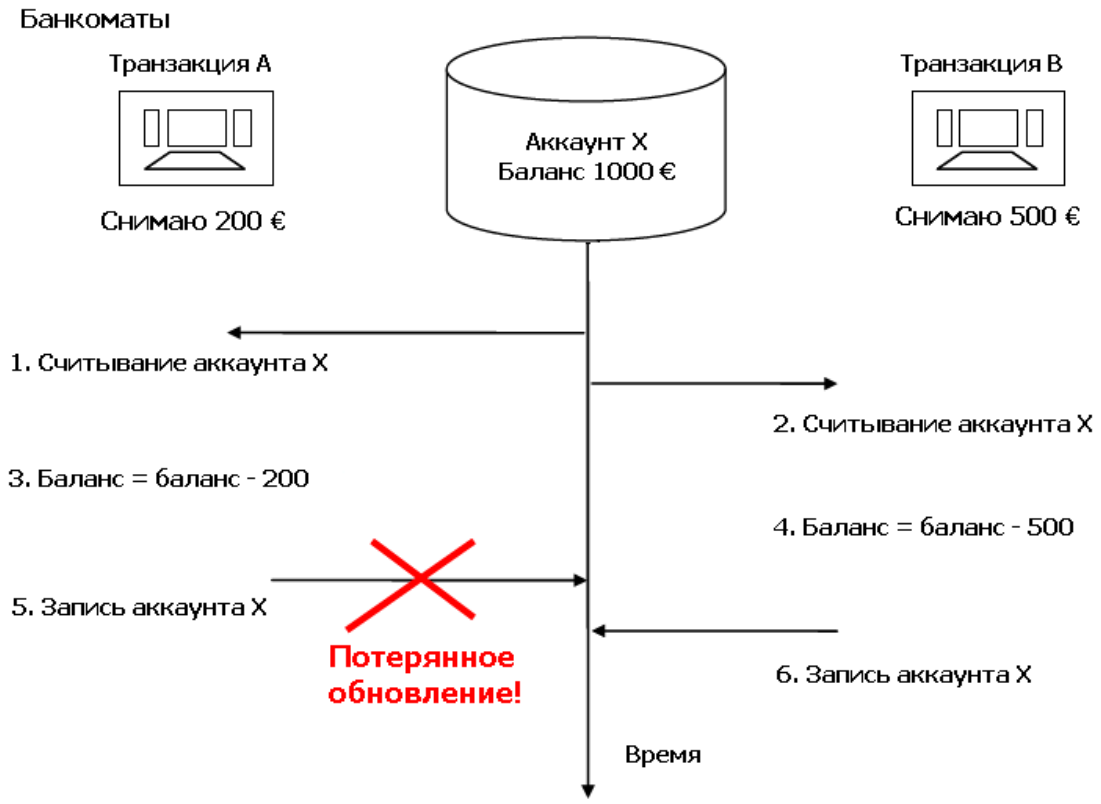


Рисунок 2.2 Проблема потерянного обновления

Без управления параллелизмом, результат записи транзакции A в шаге 5 значением 700 евро будет потерян в шаге 6, поскольку транзакция B вслепую записывает новый баланс 500 евро, который она рассчитала. Поскольку это случилось до завершения транзакции A, то явления носит название «потерянное обновление». Тем не менее, в каждом современном продукте СУБД реализован какой-то механизм управления параллелизмом, который защищает записи от перезаписывания в параллельных транзакциях до конца завершения.

Если сценарий выше реализован как последовательность «SELECT...UPDATE» и защищен схемой блокировки, то вместо потерянного обновления, сценарий переходит к DEADLOCK (что будет описано далее), в результате чего транзакция B будет понижена до прежнего уровня системой управления базами данных и транзакция A будет продолжена.

Если сценарий выше реализуется посредством **уязвимых обновлений** (на основе текущих значений), как например:

```
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 100;
```

и защищен схемой блокировки (что будет объяснено позже), то сценарий будет работать без проблем.

2.1.2 Проблема считывания «грязных» данных

Проблема считывания «грязных» данных, которая изображена на рисунке 2.3, представляет собой считывание транзакцией ненадежных (неподтвержденных) данных, которые могут ещё измениться или же в отношении обновления этих данных может быть применён откат. Такого рода транзакции не должны иметь возможности делать какие-либо обновления в базе данных, так как это привело бы к повреждению содержимого базы данных. В самом деле, любое использование неподтвержденных данных является рискованным и может привести к неправильным решениям и действиям.

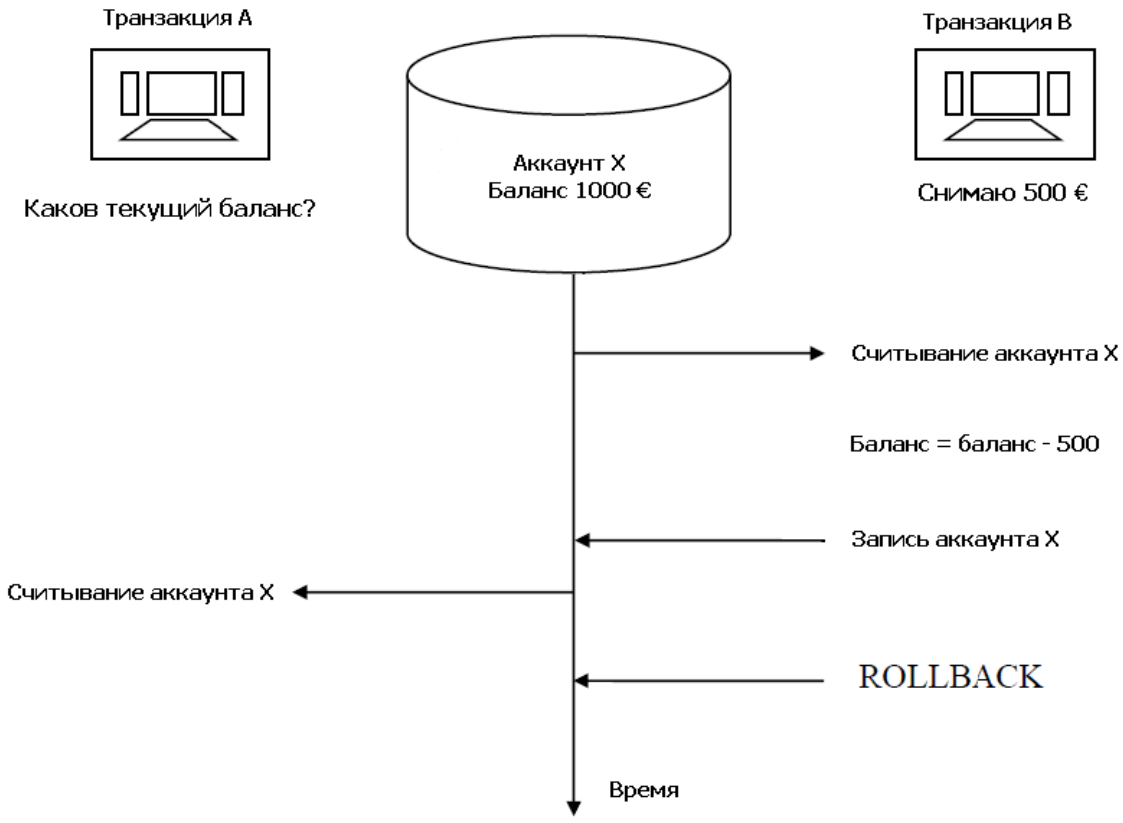


Рисунок 2.3 Пример считывания «грязных» данных

2.1.3 Проблема неповторяющегося чтения

Проблема неповторяющегося чтения, изображённая на рисунке 2.4, представляет собой нестабильность результатов запросов в транзакциях, и, если запросы должны быть повторены, то некоторые из ранее полученных строк могут быть недоступны в том виде, каком они были первоначально. Это также не исключает возможность, что в результатах повторных запросов появятся новые строки.

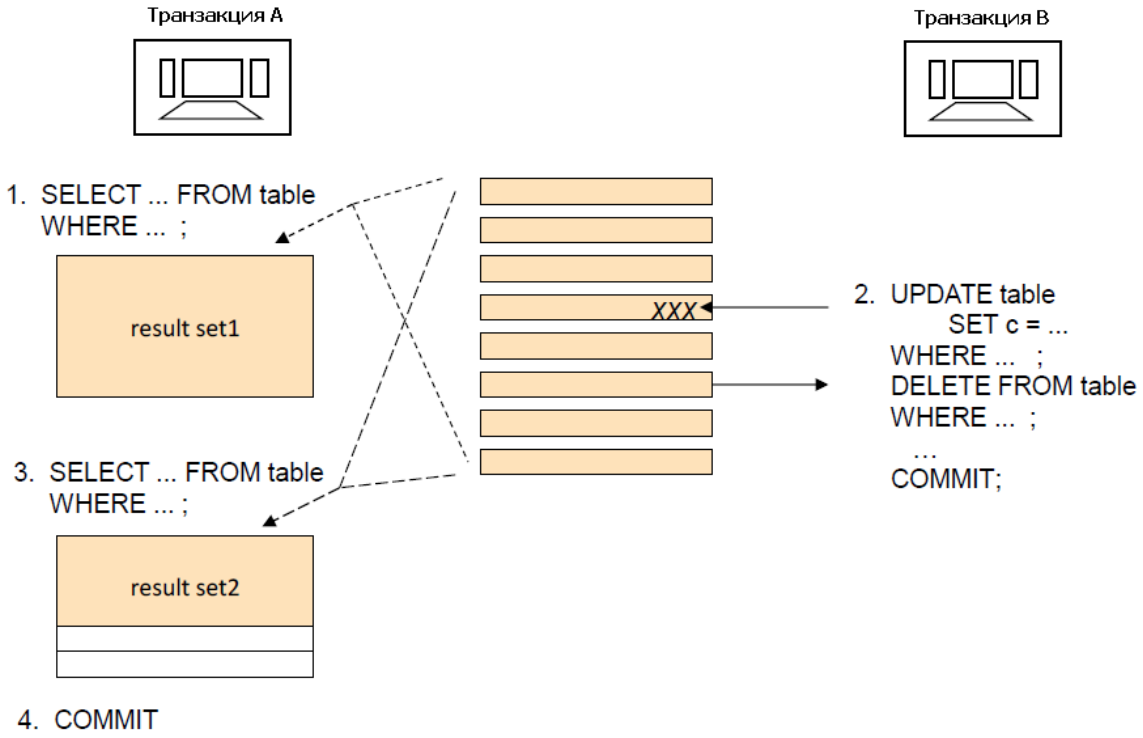


Рисунок 2.4 Проблема неповторяющегося чтения в транзакции A

2.1.4 Проблема фантомного чтения

Проблема фантомного чтения, изображённая на рисунке 2.5, означает, что результат множества запросов в транзакции может содержать новые строки, если некоторые запросы будут повторены. Они могут содержать вновь добавленные строки или же обновленные строки, в которых с целью выполнения условий поиска в запросах были изменены значения столбцов.

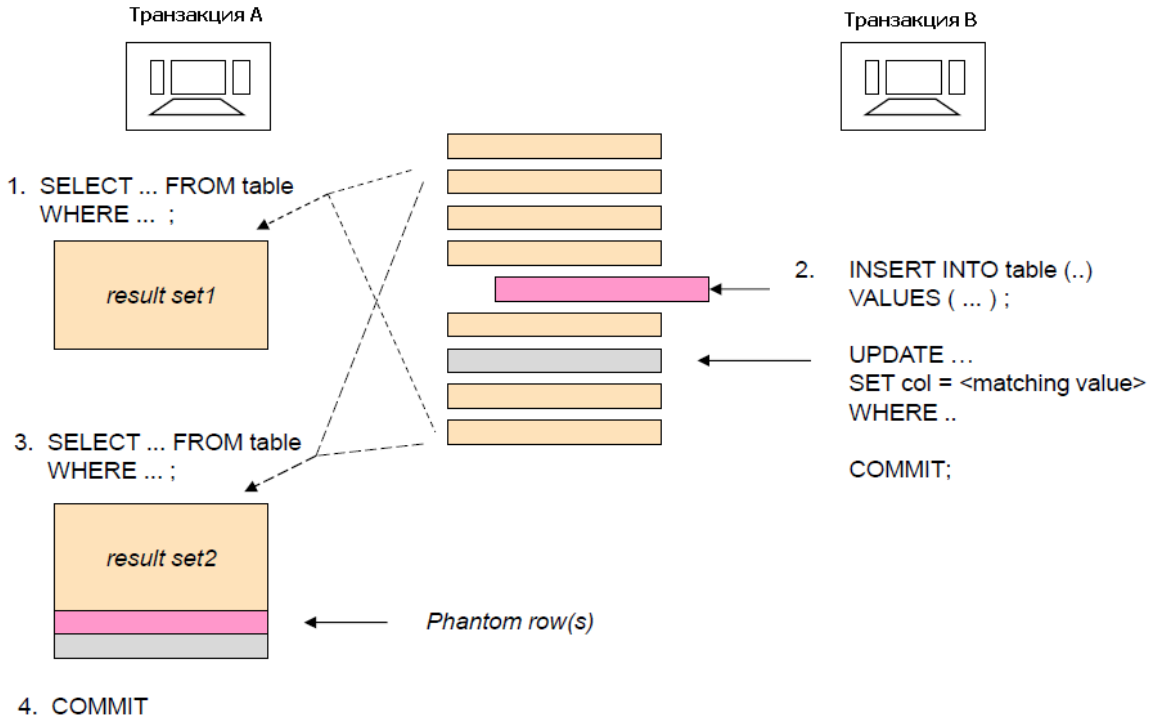


Рисунок 2.5 Пример проблемы фантомного чтения

2.2 ACID - принцип идеальной транзакции

Принцип ACID, опубликованный Тео Хардером и Андреасом Рейтером в 1983 в научном журнале ACM Computing Surveys, определяет идеал надежных транзакций SQL в многопользовательской среде. Акроним ACID происходит от сокращения названий следующих четырех свойств транзакций:

Неделимость (Atomic) Транзакция должна быть неделимой последовательностью операций («всё или ничего»), которые либо успешно выполняются до конца и фиксируются, либо выполняется откат всех этих операций.

Согласованность (Consistent) Последовательность операций будет передавать содержимое базы данных из одного согласованного состояния в другое. В момент фиксации транзакции не должно быть выявлено никаких ограничений базы данных, нарушенных операциями транзакции (первичные ключи, уникальные ключи, внешние ключи, проверки). В большинстве СУБД ограничения применяются непосредственно по отношению к каждой операции. Наше более строгое толкование последовательности требует, чтобы прикладная логика в транзакции была правильна и должным образом проверенная (правильно построенная транзакция), включая обработку исключений.

Изолированность (Isolated) Оригинальное определение Хардера и Рейтера, которое звучит как «События одной транзакции должны быть скрыты от других транзакций, выполняющихся одновременно», не может быть полностью удовлетворено большинством продуктов СУБД, которые мы объяснили в нашей статье, посвящённой параллелизму, но оно должно быть принято во внимание разработчиками приложений. Текущие продукты СУБД используют различные технологии управления параллелизмом для защиты параллельных транзакций от побочных эффектов других, поэтому разработчики приложений должны знать, как пользоваться ими должным образом.

Долговечность (Durable) Зафиксированные результаты в базе данных будут доступны на дисках несмотря на возможные сбои системы.

Принцип ACID принцип требует, чтобы транзакция, которая не соответствует этим требованиям, не должна быть зафиксирована; кроме того, приложение или сервер баз данных должны совершить откат таких транзакций.

2.3 Уровень изолированности транзакций

Свойство изолированности в принципе ACID является сложной задачей. В зависимости от механизмов управления параллелизмом, это может привести к конфликтам параллелизма и к слишком длительному времени ожидания, что снизит производительность базы данных.

ISO SQL стандарт не определяет, как должно быть реализовано управление параллелизмом, но на основе аномалий, которые мы описали выше, он определяет уровни

изолированности, которые должны решать эти проблемы согласно таблице 2.1; краткие пояснения ограничений чтения, являющихся результатом настроек этих уровней изолированности, приведены в таблице 2.2. Некоторые уровни изолированности менее жесткие, а некоторые - более строгие, что дает лучшую изолированность, но за счет возможного снижения производительности.

Следует отметить, что уровни изолированности ничего не говорят об ограничениях записи. Для операций записи необходимы, как правило, определённые блокирующие защиты, а успешно записанные данные всегда защищаются от перезаписи до конца транзакции.

Таблица 2.1 Уровни изолированности транзакций ISO SQL для устранения проблем параллелизма

Явление: / Уровень изолированности транзакций	Потерянное обновление	Считывание «грязных» данных	Неповторяющееся чтение	Фантомное чтение
READ UNCOMMITTED	НЕВОЗМОЖНО	ВОЗМОЖНО!	ВОЗМОЖНО!	ВОЗМОЖНО!
READ COMMITTED	НЕВОЗМОЖНО	НЕВОЗМОЖНО	ВОЗМОЖНО!	ВОЗМОЖНО!
REPEATABLE READ	НЕВОЗМОЖНО	НЕВОЗМОЖНО	НЕВОЗМОЖНО	ВОЗМОЖНО!
SERIALIZABLE	НЕВОЗМОЖНО	НЕВОЗМОЖНО	НЕВОЗМОЖНО	НЕВОЗМОЖНО

Таблица 2.2 Объяснение уровней изолированности транзакций ISO SQL и DB2

Уровень изолированности **ISO SQL**	Уровень изолированности **DB2**	**Объяснение изолированности**
Read Uncommitted (чтение незафиксированных данных)	UR	Позволяет чтение «грязных» незафиксированных данных, записанных конкурирующими транзакциями.
Read Committed (чтение зафиксированных данных)	CS (CC)	Не позволяет считывать незафиксированные данные конкурирующих транзакций. Oracle и DB2 (версия 9.7 и выше) будут считывать последнюю зафиксированную версию данных (в DB2 называется «Currently Committed» или сокращённо «CC» - англ. «зафиксированные на данный момент»), в то время как некоторые СУБД будут ожидать, пока данные не будут зафиксированы.
Repeatable Read (повторяющееся чтение)	RS	Позволяет считывание только зафиксированных данных, также возможно повторение счтитывния без какого-либо изменения посредством UPDATE или DELETE, внесенными параллельными

		транзакциями в строки, к которым был получен доступ.
Serializable (упорядочиваемость)	RR	Позволяет считывание только зафиксированных данных, также возможно повторение счтитывния без какого-либо изменения посредством INSERT, UPDATE или DELETE, внесенные параллельными транзакциями в таблицы, к которым был получен доступ.

Примечание 1: Обратите внимание на различие в названиях уровней изолированности в ISO SQL и DB2. В DB2 изначально было определено только 2 уровня изолированности: CS для Cursor Stability (стабильность курсора) и RR для Repeatable Read (повторяющееся чтение). Эти названия не изменялись, даже несмотря на то, что ISO SQL позднее определил 4 уровня изолированности и дал другое смысловое значение для повторяющегося чтения.

Примечание 2: В дополнение к уровням изолированности в ISO SQL, в Oracle и SQL Server был реализован уровень изолированности, называемый «**Snapshot**» (Снимок файловой системы, то есть копия файлов и директорий файловой системы на определённый момент времени). В этом случае транзакция видит снимок только зафиксированных на момент начала транзакции данных, но не видит никаких изменений в параллельных транзакциях. Тем не менее, в Oracle этот уровень называется SERIALIZABLE (упорядочиваемость).

Позже мы обсудим, какие уровни изолированности поддерживаются в СУБД, которые мы изучаем и как они реализованы. В зависимости от продукта СУБД, уровни изолированности могут быть определены как уровень базы данных по умолчанию; как уровень сессии SQL по умолчанию в начале транзакции, а в некоторых продуктах даже как подсказки исполнения запроса на уровне запросов/таблиц. Следуя передовой практике и ISO SQL, мы рекомендуем, чтобы уровень изолированности формировался в начале каждой транзакции согласно фактической потребности в изолированности этой транзакции. В соответствии со стандартом ISO SQL, Oracle и SQL Server, уровень изолированности будет установлен следующим синтаксисом команды:

```
SET TRANSACTION ISOLATION LEVEL <isolation level>
```

а вот DB2 , например, использует следующий синтаксис:

```
SET CURRENT ISOLATION = <isolation level>
```

Несмотря на разный синтаксис и названия уровеней изолированности, используемых в СУБД, ODBC и JDBC, сам API (интерфейс программирования приложений) «знает» только названия уровней изолированности ISO SQL и, например, в JDBC уровень изолированности установлен в качестве параметра подключения объекта методом **setTransactionIsolation** следующим образом:

```
<connection>.setTransactionIsolation(Connection.<transaction isolation>);
```

где `<transaction isolation>` определяется с помощью зарезервированных слов, соответствующих уровню изолированности, например, TRANSACTION_SERIALIZABLE для упорядочиваемой изолированности. Определенный уровень изолированности будет

отображаться на соответствующем уровне изолированности драйвером JDBC продукта СУБД. Если соответствующий уровень изолированности отсутствует, то SQLException будет повышен драйвером JDBC.

2.4 Механизмы управления согласованием в многопользовательской среде

Современные продукты системы управления базами данных используют для изоляции, главным образом, следующие механизмы управления параллелизмом (англ. Concurrency Control, сокращённо CC):

- Multi-Granular Locking scheme[4] (сокращённо **MGL**, также известна как **LSCC**) - схема гранулированных синхронизационных захватов;
- Multi-Versioning Concurrency Control (сокращённо **MVCC**) - многоверсионное управление параллелизмом;
- Optimistic Concurrency Control (**OCC**) - управление оптимистичным параллелизмом.

2.4.1 LSCC - схема гранулированных синхронизационных захватов

В таблице 2.3 приведены основные схемы блокировки, которые менеджер блокировок, будучи частью сервера баз данных, автоматически использует для защиты целостности данных в операциях чтения и записи. Поскольку для одной строки возможна только одна одновременная операция записи, менеджер блокировок пытается получить эксклюзивную блокировку (X-lock) для защиты строк от таких операций записи как INSERT, UPDATE или DELETE. Как видно из таблицы 2.3, X-блокировка будет предоставлена для операций записи только тогда, когда никакая другая транзакция не имеет никаких блокировок на тот же ресурс(ы), а предоставленные X-блокировки будут сохранены до конца транзакции.

Менеджер блокировок защищает целостность операций чтения, таких как SELECT, посредством совместно используемых S-блокировок, которые могут быть предоставлены для операции чтения нескольким одновременным клиентам, так как они не мешают друг другу, но это зависит от уровня изолированности транзакции. Уровень изолированности READ UNCOMMITTED (чтение незафиксированных данных) не требует S-блокировки для защиты от чтения, но в случае других уровней изолированности, для чтения необходима S-блокировка, которая будет предоставлена, если никакая другая транзакция не имеет X-блокировки на строке (строках).

[4] В литературе некоторые авторы называют схему захвата пессимистическим управлением параллелизмом (сокращённо PCC) и многоверсионным оптимистическим управлением параллелизмом (сокращённо OCC), хотя реальный OCC имеет различия в определениях параллелизма.

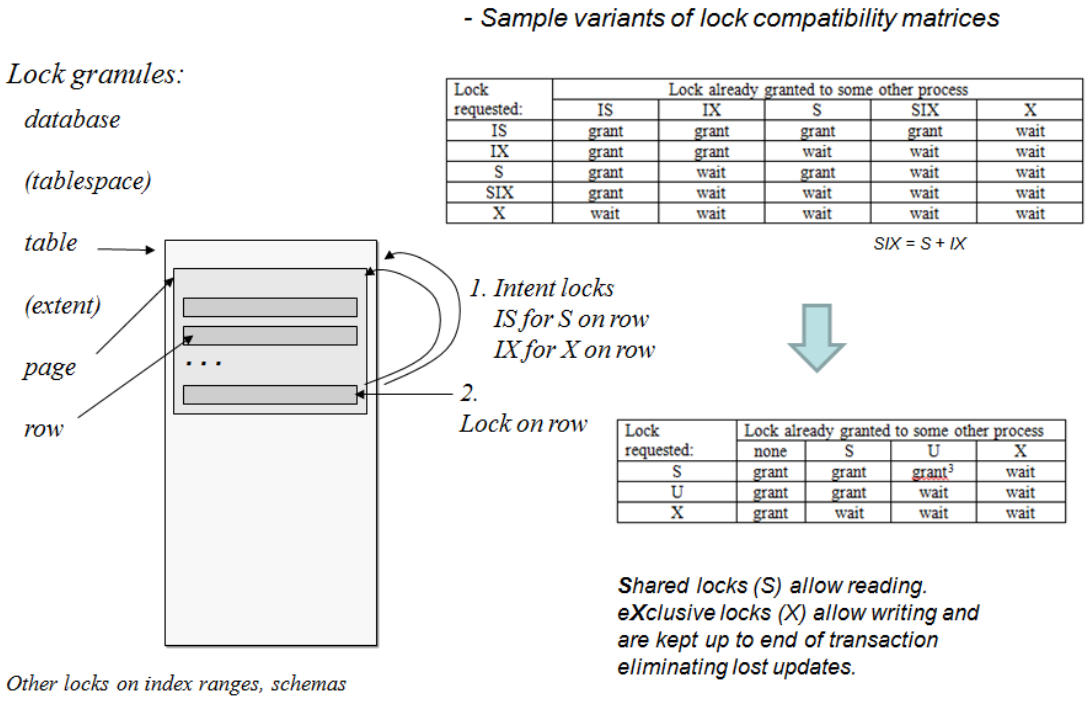
Таблица 2.3 Совместимость S-блокировок и X-блокировок

Когда транзакции необходима следующая блокировка строки	Когда другая транзакция уже имеет следующую блокировку в отношении той же строки		Когда никакая другая транзакция не имеет никаких блокировок в отношении той же строки
	S-lock	**X-lock**	
S-lock	блокировка предоставлена	ожидание разблокировки	блокировка предоставлена
X-lock	ожидание разблокировки	ожидание разблокировки	блокировка предоставлена

В случае уровня изолированности Read Committed (чтения зафиксированных данных), S-блокировка строки будет снята сразу после считывания строки, тогда как в Repeatable Read (повторяющееся чтение) и SERIALIZABLE (упорядочиваемость) S-блокировки будут сохранены до конца транзакции. Все блокировки транзакции будут сняты в конце транзакции независимо от того, как была завершена транзакция.

Примечание: В некоторых продуктах СУБД диалект SQL включает явные команды LOCK TABLE, но снимаются эти блокировки в конце транзакции всегда неявно, а в случае уровня изолированности READ COMMITTED S-блокировка снимается раньше. Неявные команды UNLOCK TABLE обычно доступны в диалектах SQL, за исключением, например, MySQL/InnoDB.

Реальные схемы блокировки СУБД значительно сложнее, они используют блокировку различных гранул, таких как строки, страницы, таблицы, диапазон индекса, схемы и т.д., а также некоторые другие способы блокировок в дополнение к имеющимся S-блокировкам и X-блокировкам. Для запросов блокировок уровня строки, менеджер блокировок сначала генерирует запрос намерения на соответствующую блокировку более высоких уровней гранул, что позволяет управлять совместимостью блокировок в схеме гранулированных синхронизационных захватов (**Multi-Granular Locking**, сокращённо **MGL**) как показано на рисунке 2.6.



Lock requested:	Lock already granted to some other process				
	IS	IX	S	SIX	X
IS	grant	grant	grant	grant	wait
IX	grant	grant	wait	wait	wait
S	grant	wait	grant	wait	wait
SIX	grant	wait	wait	wait	wait
X	wait	wait	wait	wait	wait

Lock requested:	Lock already granted to some other process			
	none	S	U	X
S	grant	grant	grant[3]	wait
U	grant	grant	wait	wait
X	grant	wait	wait	wait

Рисунок 2.6 Управление совместимостью блокировок на разных уровнях гранул

Протокол блокировки уладит проблему потерянных обновлений, но если конкурирующие транзакции используют уровень изолированности, который не снимает S-блокировки до конца транзакции, то это приведет к другой проблеме, которая показана на рисунке 2.7. Обе транзакции будут ожидать друг друга в бесконечном цикле, называемом **взаимная блокировка (Deadlock).** В ранних продуктах баз данных это было серьёзной проблемой, но современные СУБД включают в себя находящийся в спящем режиме поток выполнения, называемый **детектор взаимных блокировок (Deadlock Detector)**, который «просыпается», как правило, каждые 2 секунды (продолжительность «сна» можно изменять) для поиска взаимных блокировок, а после нахождения таковой выберет одну из ожидающих транзакций в качестве «жертвы» и произведёт этой «жертве» автоматический откат.

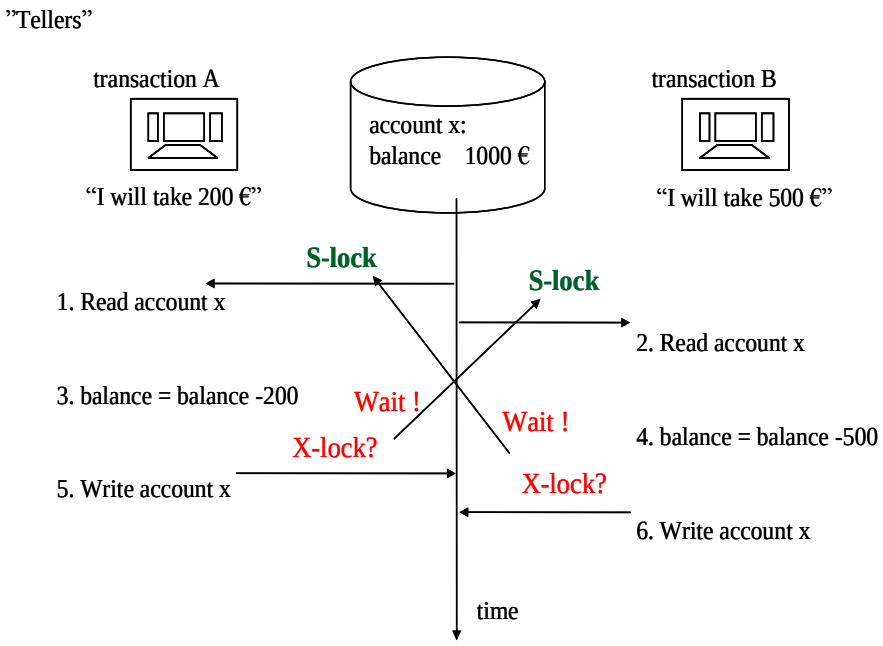


Рисунок 2.7 Проблема потерянного обновления, решённая посредством схемы гранулированных синхронизационных захватов (LSCC), но завершившаяся взаимной блокировкой

Клиентское приложение «жертвы» получит сообщение исключения по причине **взаимной блокировки** и должно будет повторить транзакцию после произвольного, но короткого времени ожидания. Смотрите часть **«retry wrapper»** (Средства программного повтора) в Java-коде примера банковского перевода (BankTransfer) в приложении 2.

> **Примечание**: Важно помнить, что **никакая СУБД не может автоматически перезапустить прерванную взаимной блокировкой «жертву»**, за это несёт ответственность код приложения или сервер приложений, на который был установлен компонент клиента доступа к данным. Важно также понимать, что взаимная блокировка не является ошибкой, а прерывание транзакции-«жертвы» - это сервис, предоставляемый сервером для того, чтобы клиентские приложения могли продолжить работу в случае, когда выполнение одновременных транзакций не может быть продолжено.

2.4.2 MVCC - многоверсионное управление параллелизмом

В MVCC техника такова, что сервер сохраняет цепь истории в некоторых метках порядка обновленных версий для данных всех строк, так что для любой обновленной строки в момент начала любой параллельной транзакции может быть найдена зафиксированная версия. Эта техника управления параллелизмом исключает время ожидания чтения и обеспечивает всего 2 уровня изолированности: любая транзакция с уровнем изолированности **READ COMMITTED** получит **последние зафиксированноые версии строк** от цепи истории, а транзакция с уровнем изолированности **SNAPSHOT** будет видеть только **последние версии строк, зафиксированные во время начала транзакции** (или записанные самой транзакцией). В уровне изолированности SNAPSHOT, транзакция никогда не увидит фантомные строки и не может даже предотвратить запись фантомных строк параллельными транзакциями, тогда как изолированность SERIALIZABLE, основанная на механизме LSCC (MGL), препятствует тому, чтобы параллельные транзакции записывали фантомные строки в базу данных. Фактически, транзакция продолжает видеть в снимке фантомные строки, которые параллельные транзакции тем временем удалили из базы данных.

Несмотря на уровень изолированности, как правило, запись также защищена в системах MVCC какой-либо блокировкой строк. Обновление строк, содержание которых тем временем был обновлено другими, будет генерировать ошибки «Your snapshot is too old» («Ваш снимок слишком старый»).

Реализации MVCC

В качестве примера внедрения MVCC, рисунок 2.8 иллюстрирует механизм MVCC, реализованный в Oracle, но он может применяться и в качестве объяснения реализации MVCC в целом. Oracle вызывает изолированность SNAPSHOT как SERIALIZABLE. Хранилище версий Oracle реализовано в Undo Tablespace. В SQL Server база данных TempDB используется в качестве хранилища версий уровня изолированности SNAPSHOT для всех баз данных этого экземпляра.

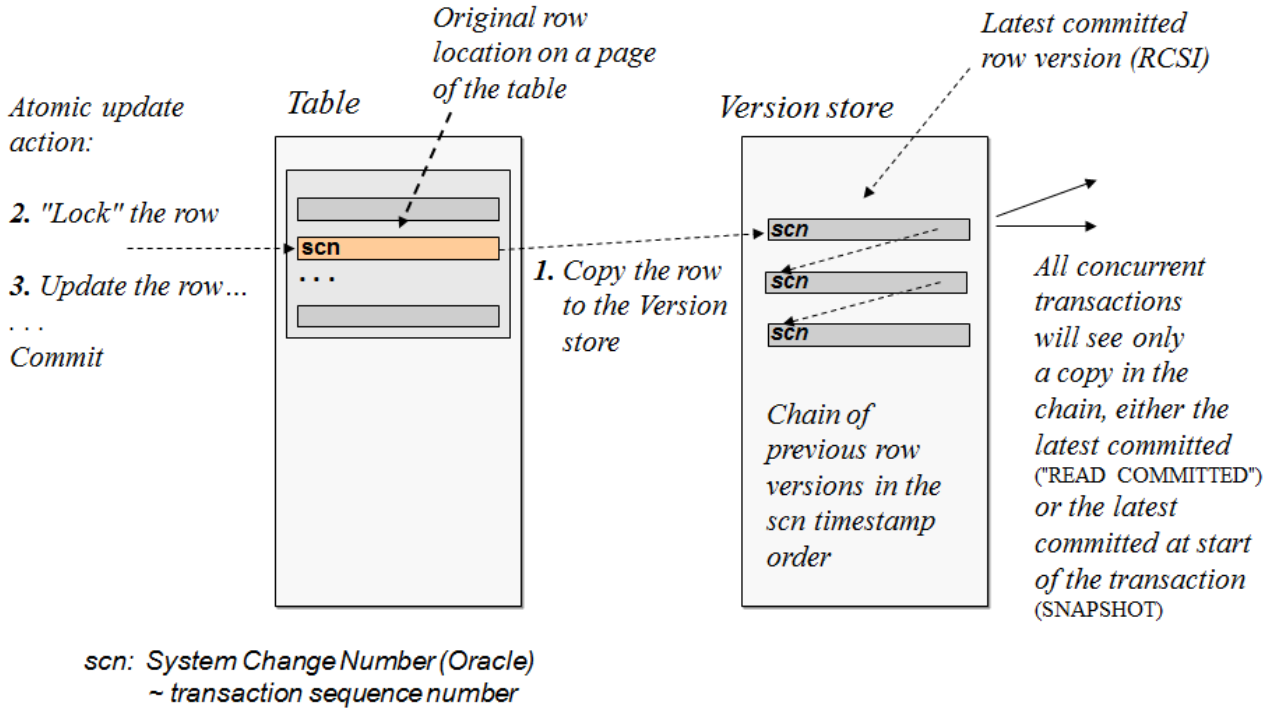


Рисунок 2.8 Многоверсионное управление параллелизмом (MVCC), основанное на цепи истории зафиксированных строк

В MVCC Oracle первой транзакции для записи строки (то есть для выполнения вставки, обновления или удаления) будет предоставлена блокировка строки и приоритет записи, а конкурирующие записи будут помещены в очередь. Блокировки строки реализуются при помощи маркировки записанных строк посредством SCN (англ. «System Change Number» - системный номер изменения) транзакции, порядковыми номерами начатой транзакции. Поскольку SCN строки принадлежит активной транзакции, то эта строка будет зарезервирована для этой транзакции. Использование блокировки по записи означает, что взаимные блокировки возможны, но вместо автоматического прерывания «жертвы», Oracle немедленно находит блокировку строки, которая бы могла привести к взаимной блокировке, вызывает исключение в приложении клиента и ожидает клиента, чтобы разрешить взаимную блокировку явной командой отката (ROLLBACK).

Технику управления параллелизмом в Oracle можно назвать гибридным CC, так как в дополнение к MVCC с неявной блокировкой строки, Oracle обеспечивает явные команды «LOCK TABLE», а также явную блокировку строк посредством команды «SELECT ... FOR UPDATE», которая обеспечивает средства для предотвращения невидимых фантомных строк. В Oracle транзакция также может быть объявлена как Read Only (только для чтения).

Microsoft также отметил преимущества MVCC, а в SQL Server, начиная с версии 2005, стало возможным настроить в сервере использование управления версиями строк при помощи настроек свойств базы данных посредством команд Transact-SQL, а начиная с версии 2012 – посредством свойств базы данных, как это показано на рисунке 2.9.

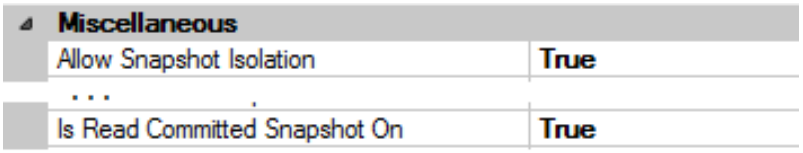


Рисунок 2.9 Настройка SQL Server 2012 для поддержки Snapshots

Механизм управления параллелизмом в MySQL/InnoDB является реальным гибридным CC, обеспечивающим для чтения четыре уровня изолированности:

- READ UNCOMMITTED считывает строки таблицы без S-блокировки;
- READ COMMITTED (фактически «Read Latest Committed») считывает строки таблицы даже когда они заблокированы посредством MVCC;
- REPEATABLE READ (фактически «Snapshot»), использующий MVCC;
- SERIALIZABLE, использующий MGL CC с S-блокировками для предотвращения появления фантомных строк.

Примечание: независимо от уровня изолированности запись всегда будет нуждаться в защите исключительной блокировкой.

2.4.3 OCC - управление оптимистичным параллелизмом

В оригинальной OCC все изменения, внесенные транзакцией хранятся отдельно от базы данных и синхронизируются с базой данных только в фазе фиксации (COMMIT). Это было реализовано, например, в СУБД Pyrrho Университета Западной Шотландии. Единственным и неявным уровнем изолированности, доступном в СУБД Pyrrho, является SERIALIZABLE (**http://www.pyrrhodb.com**).

2.4.4 Резюме

Разработчиком стандарта ISO SQL является ANSI и изначально был основан на версии SQL-языка DB2, которую IBM уступила ANSI. Механизм управления параллелизмом в DB2 основан на схеме MGL (Multi-Granular Locking), на тот момент у DB2 было только 2 уровня изолированности: Cursor Stability (CS) и Repeatable Read (RR). Очевидно, что это имело влияние на семантику новых уровней изолированности (см. таблицу 2.2 в пункте 2.3), определенных для ANSI/ISO SQL, которые могут быть поняты с точки зрения механизмов блокировки.

В стандарте SQL ничего не говорится о реализациях, и как мы видели в предыдущих пунктах, были применены и другие механизмы параллелизма, а те же названия уровней изолированности были использованы в несколько иной интерпретации. Под названием «Уровни изолированности» в таблице 2.4 уровни на синем фоне «Read Uncommitted», «Read Committed», «Repeatable Read» и «Serializable» представляют собой уровни изолированности в том виде, как мы их понимаем в стандартной семантике SQL. Одинаковые названия в кавычках в столбцах продуктов, например «serializable», указывают на семантику, отличающуюся от уровней изолированности стандарта. Название уровня изолированности «read latest committed» изобретено нами, поскольку в различных СУБД используются различные названия уровней изолированности, в которых операции чтения проходят без необходимости ожидания блокировки либо текущей строки или последней зафиксированной версии строки в случае, если строка была обновлена, но еще не была зафиксирована какой-либо параллельной транзакцией. Эти семантики, основанные на снимке (snapshot), вызвали замешательство, и понятиям изолированности, используемым в стандарте, возможно, понадобились бы разъяснения и дополнения. Группа разработчиков стандарта ANSI/SQL и авторов учебника базы данных признала наличие проблемы в своей статье "A Critique of ANSI SQL Isolation Levels" (Беренсон и др., 1995).

Выполнение упражнения 2.7 в практических лабораторных заданиях с использованием различных СУБД показывает некоторые проблемы снимков (snapshot) в случае записи в базу данных, так как запись будет нарушать согласованность снимка (snapshot) текущего состояния. Следовательно, безопасное использование снимка (snapshot) возможно только для получения отчетов. В таблице 2.4 приведены и некоторые другие различия между СУБД.

Таблица 2.4 Поддерживаемые функциональные возможности транзакций в стандарте ISO SQL и продуктах СУБД

	ANSI/ISO SQL	DB2	Oracle	SQL SERVER	MySQL/InnoDB	PostgreSQL	Pyrrho
	SQL:2006	LUW 9.7	12g1	2012	5.6	9.2	4.8
autocommit (server-side)	n/a	n/a	n/a	yes	yes	yes	yes
Transaction Limits							
explicit start	yes	n/a	n/a	yes	yes	yes	yes
implicit start	yes	yes	yes	(configurable)	(configurable)	n/a	n/a
COMMIT	yes	yes	yes	yes	yes	yes	yes
implicit commit on DDL	n/a	n/a	yes	n/a	yes	n/a	n/a
ROLLBACK	yes	yes	yes	yes	yes	yes	yes
implicit rollback on concurrency conflict (deadlock)	yes	yes	no (exception raised)	yes	yes	no (transaction invalidated)	yes, at commit
implicit rollback on error	implementation dependent	n/a	n/a	(configurable)	n/a	no (transaction invalidated)	yes
SAVEPOINT	yes	yes	yes	yes	yes	yes	n/a
ROLLBACK TO SAVEPOINT	yes	yes	yes	yes	yes	yes	n/a
RELEASE SAVEPOINT	yes	yes	yes	n/a	yes	yes	n/a
Isolation levels							
READ UNCOMMITTED	yes	UR	n/a	yes	yes	n/a (1)	n/a
"read latest committed"	n/a	CS (currently committed)	"read committed"	(configurable)	"read committed"	"read committed"	n/a
READ COMMITTED	yes	CS	n/a	yes	n/a	n/a (2)	n/a
REPEATABLE READ	yes	RS	n/a	yes	n/a	n/a (2)	n/a
snapshot		n/a	"serializable"	(configurable)	"repeatable read"	"serializable"	"serializable"
SERIALIZABLE	yes	RR	explicit locking	yes	yes	explicit locking	"serializable"
note: isolation levels in upper-case stand for ISO/SQL semantics						(1) migrate to "read latest committed"	
						(2) migrate to snapshot	

Снимок означает согласованное отображение базы данных в начале транзакции. По существу это отлично подходит для транзакций «только для чтения», так как в этом случае эта транзакция не блокирует параллельные транзакции.

Семантика снимка (snapshot) не исключают существования фантомов, так как они просто не включены в снимок. В продуктах ISO SQL с использованием MVCC семантики «Serializable» предотвращают существование фантомов и могут быть реализованы с использованием надлежащего обеспечения блокировок на уровне таблиц, например, в Oracle и PostgreSQL. В случае изолированности снимка все продукты допускают команду INSERT, но возможности других операций, допускающих запись, могут отличаться.

Из таблицы 2.4 видно, что DB2 является единственным продуктом в нашей СУБД DebianDB, который не поддерживает изолированность снимка.

2.5 Практические лабораторные задания

Новая пользовательская сессия MySQL начата обычным способом (рисунок 2.10):

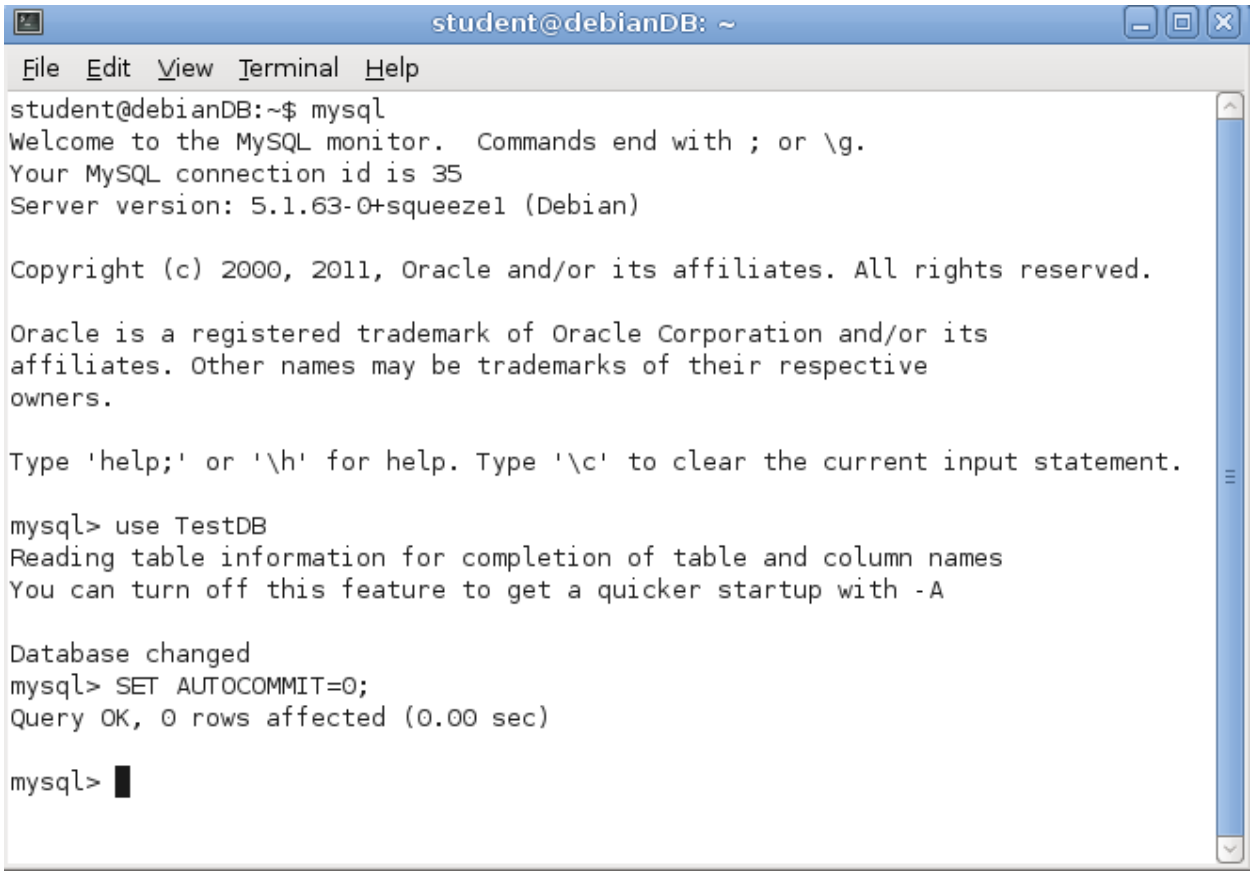


Рисунок 2.10 Начало пользовательской сессии MySQL

УПРАЖНЕНИЕ 2.0

Для тестов параллелизма нам нужно второе окно терминала и новая сессия MySQL, как показано на рисунке 2.10. Пусть сессия в левом окне терминала соответствуют сессии A, сессия в правом окне терминала соответствуют сессии B. Обе сессии начинаются с подключения к базе данных TestDB и отключения режима автоматической фиксации AUTOCOMMIT:

```
-----------------------------------
use TestDB
SET AUTOCOMMIT = 0;
-----------------------------------
```

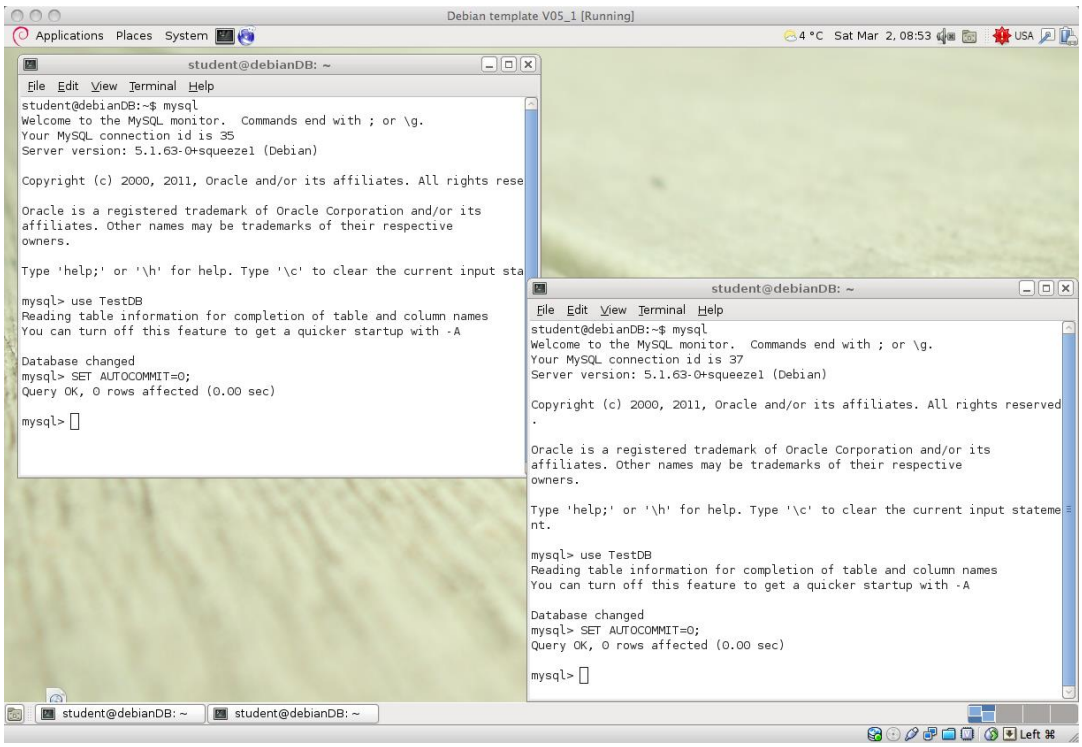


Рисунок 2.11 Параллельные сессии MySQL в виртуальной машине DBTechNet

Как мы видим, параллельные транзакции, получающие доступ к тем же самым данным, могут блокировать другие. Таким образом, транзакции должны быть разработаны так, чтобы быть как можно более короткими, только для выполнения необходимой работы. Включение диалога с конечным пользователем в логике транзакции SQL приведет к катастрофическому времени ожидания в производственной среде. Поэтому **необходимо, чтобы никакие транзакции SQL не передавали управление пользовательскому уровню интерфейса прежде, чем транзакция закончится.**

Текущие настройки уровня изолированности проверяются следующим образом при помощи системных переменных в команде SELECT:

```
-----------------------------------
SELECT @@GLOBAL.tx_isolation, @@tx_isolation;
-----------------------------------
```

Видно, что в MySQL/InnoDB уровнем изолированности по умолчанию является REPEATABLE READ, как на глобальном, так и на местном уровне (сессии).

Для обеспечения безопасности таблица аккаунтов удаляется и создаётся заново/инициализируется для регистрации двух строк данных:

```
-----------------------------------
DROP TABLE Accounts;
CREATE TABLE Accounts (
acctID INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL,
CONSTRAINT remains_nonnegative CHECK (balance >= 0)
);
SET AUTOCOMMIT = 0;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

В соответствии с ISO SQL уровень изолированности транзакции должен быть установлен в начале транзакции и в транзакции он не может быть изменен позже. Поскольку реализации могут различаться, мы проверим на следующем примере, когда уровень изолированности транзакций может быть установлен в MySQL / InnoDB в случае явных транзакций:

```
-----------------------------------
START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;
-- Then another try in different order:
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;
-----------------------------------
```

Согласно ISO SQL, в уровне изолированности READ UNCOMMITTED невозможно применить действие записи, поэтому давайте проверим поведение MySQL в этом случае:

```
-----------------------------------
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
DELETE FROM Accounts;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;
-----------------------------------
```

Вопрос

- Какой вывод (какие выводы) можно сделать?

УПРАЖНЕНИЕ 2.1

Как указывалось выше, проблема потерянных обновлений подразумевает перезапись только что обновлённого значения строки другой (параллельной) транзакцией ДО окончания данной транзакции. Любой современный продукт СУБД с сервисами управления параллелизмом предотвращает эту проблему, поэтому её воспроизведение в тестах невозможно. Однако ПОСЛЕ фиксации транзакции любая НЕОСТОРОЖНАЯ параллельная транзакция может переписать результаты без первого считывания текущего

зафиксированного состояния. В этом случае мы имеем дело со «слепой перезаписью» (Blind Overwriting) или «записью «грязных» данных» (Dirty Write), воспроизведение которых может оказаться даже слишком простым.

Ситуация со «слепой перезаписью» возникает, например, когда приложение считывает значения в базе данных, обновляет значения в памяти, а затем записывает обновленное значение в базу данных. В следующей таблице мы моделируем часть приложения, используя местные переменные в MySQL. Местная переменная – это просто именной заполнитель в рабочей памяти текущей сессии. Он идентифицирован символом @, добавленным к началу его имени, и он может быть включен в синтаксис операторов SQL, при условии, что он всегда принимает единственное скалярное значение.

Таблица 2.4 предлагает поэкспериментировать с параллельными сессиями A и B (каждая в отдельном столбце). Порядок операторов SQL для выполнения этого упражнения отмечен в первом столбце. Цель состоит в том, чтобы смоделировать ситуацию потерянных обновлений, изображенной на рисунке 2.2: транзакцией в сессии A является списание 200 € со счета 101 и транзакцией в сессии B является списание 500 € с того же счета, что приведет к перезаписи значения баланса счета (потеря информации транзакцией A о снятии 200 €). Таким образом, результат «слепой перезаписи» будет такой же, как если бы это был результат потерянных обновлений.

Чтобы содержимое базы данных не зависело от предыдущих экспериментов, прежде всего нужно восстановить исходное содержание таблицы счетов следующим образом:

```
-----------------------------------
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Таблица 2.4 Проблема «слепой перезаписи», приложение моделировано при помощи местных переменных

	Session A	Session B
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** -- Amount to be transfered by A SET @amountA = 200; SET @balanceA = 0; -- Init value SELECT balance INTO @balanceA FROM Accounts WHERE acctID = 101; SET @balanceA = @balanceA - @amountA; SELECT @balanceA;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;**

		`-- Amount to be transfered by B`
		`SET @amountB = 500;`
		`SET @balanceB = 0; -- Init value`
		
		`SELECT balance INTO @balanceB`
		`FROM Accounts WHERE acctID = 101;`
		
		`SET @balanceB = @balanceB - @amountB;`

3	`UPDATE Accounts`
	`SET balance = @balanceA`
	`WHERE acctID = 101;`	

4		`UPDATE Accounts`
		`SET balance = @balanceB`
		`WHERE acctID = 101;`

5	`SELECT acctID, balance`
	`FROM Accounts`
	`WHERE acctID = 101;`
	`COMMIT;`	

6		`ELECT acctID, balance`
		`FROM Accounts`
		`WHERE acctID = 101;`
		`COMMIT;`

Примечание: На четвёртом шаге следует иметь в виду, что по умолчанию продолжительность блокировки в MySQL равна 90 секундам. Таким образом, транзакция (клиент) A должны перейти к пятому шагу без задержки, сразу после четвёртого шага транзакции (клиента) B.

Учитывая параллельное (чередующееся, если быть более точным) выполнение транзакций A и B в таблице 2.4, A готовится вывести 200 € с банковского счета в первом шаге (пока без обновления базы данных), B готовится вывести 500 € во втором шаге, A обновляет базу данных в третьем шаге, B обновляет базу данных в четвёртом шаге, A проверяет баланс банковского счета с целью убедиться в его соответствии ожидаемому значению перед фиксацией в пятом шаге, B делает то же самое в шестом шаге.

Вопросы

- Было ли поведение системы ожидаемым?
- Есть ли в этом случае доказательства присутствия потерянных данных?

Примечание: Все продукты СУБД осуществляют управление параллелизмом, поэтому ни в каком уровне изолированности аномалия потерянных обновлений никогда не проявится. Однако, всегда есть вероятность, что небрежно написанный код приложения станет причиной появления «слепой перезаписи», последствия которой будут иметь такие же катастрофические последствия с аномалией обновления. По сути дела, это похоже на аномалию потерянного обновления, переходящую в сценарий одновременно выполняемых операций через чёрный ход (back door).

УПРАЖНЕНИЕ 2.2a

Повторение упражнения 2.1, но уже с использованием уровня изолированности SERIALIZABLE для MGL (Multi-Granular Locking).

Сначала восстановим исходное содержание таблицы счетов:

```
-----------------------------------
SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Таблица 2.5а Сценарий упражнения 2.1 с удержанием S-блокировок

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL SERIALIZABLE;** -- Amount to be transfered by A SET @amountA = 200; SET @balanceA = 0; -- Init value SELECT balance INTO @balanceA FROM Accounts WHERE acctID = 101; SET @balanceA = @balanceA - @amountA; SELECT @balanceA;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL SERIALIZABLE;** -- Amount to be transfered by B SET @amountB = 500; SET @balanceB = 0; -- Init value SELECT balance INTO @balanceB FROM Accounts WHERE acctID = 101; SET @balanceB = @balanceB - @amountB;
3	UPDATE Accounts SET balance = @balanceA WHERE acctID = 101;	
4		-- continue without waiting for A! UPDATE Accounts SET balance = @balanceB WHERE acctID = 101;
5	SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**	

6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**

Вопросы

a) Какой вывод (какие выводы) можно сделать?

b) Что произойдёт, если в обеих транзакциях «SERIALIZABLE» будет заменён на «REPEATABLE READ»?

> **Примечание**: MySQL / InnoDB реализует REPEATABLE READ посредством MVCC для считывания, а SERIALIZABLE посредством схемы блокировок MGL.

УПРАЖНЕНИЕ 2.2b

Повторение упражнения 2.2a, на этот раз с использованием уязвимых обновлений (sensitive updates) в сценарии SELECT – UPDATE без локальных переменных.

Сначала восстановим исходное содержание таблицы счетов:

```
-----------------------------------
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Таблица 2.5b Конкуренция SELECT – UPDATE с использованием уязвимых обновлений

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL SERIALIZABLE;** SELECT balance FROM Accounts WHERE acctID = 101;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL SERIALIZABLE;** SELECT balance FROM Accounts WHERE acctID = 101;
3	UPDATE Accounts SET balance = balance – 200 WHERE acctID = 101;	
4		UPDATE Accounts SET balance = balance – 500 WHERE acctID = 101;

5	SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**	
6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**

Вопрос

- Какой вывод (какие выводы) можно сделать?

<u>УПРАЖНЕНИЕ 2.3</u> Конкуренция UPDATE – UPDATE на двух ресурсах в различном порядке

Сначала восстановим исходное содержание таблицы счетов:

```
-----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Таблица 2.6 Сценарий UPDATE-UPDATE

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** UPDATE Accounts SET balance = balance - 200 WHERE acctID = 202;
3	UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;	
4		UPDATE Accounts SET balance = balance + 200 WHERE acctID = 101;
5	**COMMIT;**	

6		**COMMIT;**

Вопрос

- Какой вывод (какие выводы) можно сделать?

> **Примечание**: Уровень изолированности транзакции не играет никакой роли в этом сценарии, но это хорошая практика, чтобы всегда определять уровень изоляции в начале каждой транзакции! Там могут быть некоторые скрытые обработки, например, проверки с помощью внешнего ключа и триггеры, использующие доступ считывания. На самом деле за конструкцию триггеров должны нести ответственность администраторы баз данных, что выходит за рамки этого руководства.

УПРАЖНЕНИЕ 2.4 (Считывание «грязных» данных - Dirty Read)

To continue with the transaction anomalies, an attempt is now made to produce the occurrence of a dirty read situation. Transaction A runs in (MySQL's default) REPEATABLE READ, whereas transaction B is set to run in READ UNCOMMITTED isolation level:

В продолжение экспериментов с аномалиями попытаемся смоделировать возникновение ситуации считывания «грязных» данных. Транзакция A осуществляется в REPEATABLE READ (по умолчанию в MySQL), тогда как транзакция B настроена для работы в режиме уровня изолированности READ UNCOMMITTED:

Сначала восстановим исходное содержание таблицы счетов:

```
-----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Таблица 2.7 Проблема считывания «грязных» данных

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL REPEATABLE READ;** UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101; UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ UNCOMMITTED;**

		SELECT * FROM Accounts; **COMMIT;**
3	**ROLLBACK;** SELECT * FROM Accounts; **COMMIT;**	

Вопросы

a) Какой вывод (какие выводы) можно сделать? Что мы можем сказать о надежности транзакции B?
b) Как мы можем решить эту проблему?

УПРАЖНЕНИЕ 2.5 (Проблема неповторяющегося чтения – Non-Repeatable Read)

Далее следует аномалия неповторяющегося чтения:

Сначала восстановим исходное содержание таблицы счетов:

```
-----------------------------------
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Таблица 2.8 Проблема неповторяющегося чтения

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** SELECT * FROM Accounts WHERE balance > 500;	
2		SET AUTOCOMMIT = 0; UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101; UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202; SELECT * FROM Accounts; **COMMIT;**
3	-- Repeating the same query SELECT * FROM Accounts WHERE balance > 500; **COMMIT;**	

Вопросы

a) Считывает ли транзакция A в третьем шаге тот же результат, что она считала в первом шаге?

b) Как насчет установки транзакции в уровень изолированности REPEATABLE READ?

УПРАЖНЕНИЕ 2.6

Теперь предпримем попытку воспроизведения классической «вставки фантома» из учебников:

Сначала восстановим исходное содержание таблицы счетов:

```
-----------------------------------
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Таблица 2.9 Проблема «вставки фантома»

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`ISOLATION LEVEL REPEATABLE READ;`
	`START TRANSACTION READ ONLY;`	

2		`SET AUTOCOMMIT = 0;`
		`INSERT INTO Accounts (acctID,`
		`balance) VALUES (301,3000);`
		`COMMIT;`

3	`SELECT * FROM Accounts` `WHERE balance > 1000;`	

4		`INSERT INTO Accounts (acctID,`
		`balance) VALUES (302,3000);`
		`COMMIT;`

5	`-- Can we see accounts 301 and 302?`
	`SELECT * FROM Accounts`
	`WHERE balance > 1000;`
	
	`COMMIT;`	

Вопросы

a) Должна ли транзакция B ожидать транзакцию A в каждом шаге?

b) Являются ли недавно вставленные номера аккаунтов 301 и/или 302 видимыми в среде транзакции A?

c) Влияет ли это на результаты четвёртого шага, если мы изменим порядок шагов 2 и 3?

d) (вопрос повышенного уровня сложности)
MySQL / InnoDB использует многоверсионность для изолированности REPEATABLE READ, но что является надлежащей меткой снимка: время начала транзакции или первой команды SQL? Обратите внимание, что даже в режиме транзакций мы можем использовать команду START TRANSACTION (и установить некоторые свойства транзакции, как это сделано в данном примере путём объявления транзакции «только для чтения»).

Задача: изучить вопрос о предупреждении появления фантомов путём замены уровня изолированности REPEATABLE READ на SERIALIZABLE.

УПРАЖНЕНИЕ 2.7 Исследование SNAPSHOT с различными видами фантомов

Для начала запустим новое окно терминала для "Клиента C" (которое в нашем тесте мы будем использовать и далее)

```
-----------------------------------
mysql TestDB
SET AUTOCOMMIT = 0;
DROP TABLE T;
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(40), i SMALLINT);
INSERT INTO T (id, s, i) VALUES (1, 'first', 1);
INSERT INTO T (id, s, i) VALUES (2, 'second', 2);
INSERT INTO T (id, s, i) VALUES (3, 'third', 1);
INSERT INTO T (id, s, i) VALUES (4, 'fourth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);
COMMIT;
-----------------------------------
```

Таблица 2.10 Проблемы фантомов Insert и Update и «фантомное обновление» удалённой строки

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL REPEATABLE READ;** SELECT * FROM T WHERE i = 1;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** UPDATE T SET s = **'Update by B'** WHERE id = 1; INSERT INTO T (id, s, i) VALUES (6, **'Insert Phantom'**, 1); UPDATE T SET s = **'Update Phantom'**, i = 1 WHERE id = 2;

		DELETE FROM T WHERE id = 5; SELECT * FROM T;
3	-- Repeat the query and do updates SELECT * FROM T WHERE i = 1; INSERT INTO T (id, s, i) VALUES (7, **'inserted by A'**, 1); UPDATE T SET s = **'update by A inside the snapshot'** WHERE id = 3; UPDATE T SET s = **'update by A outside the snapshot'** WHERE id = 4; UPDATE T SET s = **'update by A after update by B'** WHERE id = 1;	
3.5		**-- Client C:** **-- what's the current content?** SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; SELECT * FROM T;
4		**-- Client B continues** **-- without waiting for A** **COMMIT;** **SELECT * FROM T;**
5	SELECT * FROM T WHERE i = 1; UPDATE T SET s = **'updated after delete?'** WHERE id = 5; SELECT * FROM T WHERE i = 1; **COMMIT;**	
6	SELECT * FROM T; **COMMIT;**	
7		-- Client C does the final select SELECT * FROM T; COMMIT;

Вопросы

a) Являются ли insert (вставка) и update (обновление), выполненные транзакцией B видимыми в среде транзакции A?
b) Что происходит когда транзакция A пытается обновить строку 1, обновленную транзакцией B?
c) Что происходит когда транзакция A пытается обновить строку с id=5, удалённую транзакцией B?

> **Примечание**: Операция SELECT в транзакциях MySQL/InnoDB, которая работает с использованием уровня изолированности REPEATABLE READ, создает согласованный снимок. Если транзакция манипулирует строками в базовой таблице (базовых таблицах) снимка, то такой снимок больше не является согласованным.

Перечень 2.1 Примеры результатов выполнения теста упражнения 2.7

```
mysql> -- 5. Client A continues
mysql> SELECT * FROM T WHERE i = 1;
+----+-----------------------------+------+
| id | s                           | i    |
+----+-----------------------------+------+
|  1 | update by A after B         |    1 |
|  3 | update by A inside snapshot |    1 |
|  5 | to be or not to be          |    1 |
|  7 | inserted by A               |    1 |
+----+-----------------------------+------+
4 rows in set (0.01 sec)

mysql> UPDATE T SET s = 'updated after delete?' WHERE id = 5;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> SELECT * FROM T WHERE i = 1;
+----+--------------------------------+------+
| id | s                              | i    |
+----+--------------------------------+------+
|  1 | update by A after update by B |    1 |
|  3 | update by A inside snapshot   |    1 |
|  5 | to be or not to be            |    1 |
|  7 | inserted by A                 |    1 |
+----+--------------------------------+------+
4 rows in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> -- 6. Client A continues with a new transaction
mysql> SELECT * FROM T;
+----+-------------------------------+------+
| id | s                             | i    |
+----+-------------------------------+------+
|  1 | update by A after update by B |    1 |
|  2 | Update Phantom                |    1 |
|  3 | update by A inside snapshot   |    1 |
|  4 | update by A outside snapshot  |    2 |
|  6 | Insert Phantom                |    1 |
|  7 | inserted by A                 |    1 |
+----+-------------------------------+------+
6 rows in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Примечание: В конце приложения 1 для сравнения приведены результаты упражнения 2.7 из SQL Server 2012.

3 Несколько хороших советов

Пользовательская транзакции обычно нуждается в многократных диалогах с базой данных. Некоторые из этих диалогов только собирают данные из базы данных, способствующие пользовательской транзакции, и, как заключительный этап пользовательской транзакции, кнопка «Сохранить» вызовет транзакцию SQL, которая обновит базу данных.

Операции SQL, даже в той же последовательности транзакций пользователя, могут иметь различную надежность и требования к изолированности. Всегда определяйте уровень изолированности в начале каждой транзакции.

Согласно стандарту SQL уровень изолированности READ UNCOMMITTED может быть использован только в режиме READ ONLY (Мелтон и Саймон 2002), но сами продукты СУБД не принуждают к этому.

Продукты СУБД отличаются друг от друга управлением параллельным выполнением сервисов и управлением поведением транзакции, а для надежности и производительности важно, чтобы разработчик приложения знал поведение СУБД, которая будет использоваться.

Надежность является приоритетом номер один, даже важнее производительности и других показателей, но уровень изолированности транзакций по умолчанию, используемый СУБД, часто оказывается впереди надежности. Должный уровень изолированности должен планироваться с особой осторожностью, если разработчик не может решить, какой уровень изолированности является достаточно надежным, то в этом случае должен быть использован уровень изолированности SERIALIZABLE с семантикой ISO SQL. Важно понимать, что уровень изолированности SNAPSHOT только гарантирует согласованность результатов, но не сохраняет содержимое базы данных. Если уровни изолированности в вашей СУБД поддерживают только SNAPSHOT, а фантомы в вашем случае недопустимы, то тогда необходимо изучить возможности использования явной блокировки.

Транзакции SQL не должны содержать никаких диалогов с конечным пользователем, поскольку это замедлило бы обработку. Так как транзакции SQL могут быть понижены до прежнего уровня во время транзакции, то они не должны затрагивать ничего, кроме самой базы данных. Транзакция SQL должна быть максимально короткой, чтобы минимизировать конкуренцию параллелизма и блокирование параллельных транзакций.

Избегайте DDL команд в транзакциях. Неявные фиксации из-за DDL команд могут привести к неумышленным транзакциям.

Каждая транзакция SQL должна иметь четко определенные задачи, начинаясь и заканчиваясь в том же компоненте приложения. В этой обучающей программе мы не будем рассматривать тему использования SQL транзакций в хранимых процедурах, так как хранимые процедуры языков и их реализации различны в различных СУБД. Некоторые из этих продуктов СУБД в нашей лаборатории DebianDB не допускают наличия в хранимых процедурах оператора COMMIT. Тем не менее, транзакция может вынужденно получить откат даже в середине некоторых хранимых процедур, и это также должно быть обработано в вызывающем коде приложения.

Техническим контекстом транзакции SQL является наличие единственного соединения с базой данных. Если транзакция не удается из-за конфликта параллелизма, то в большинстве случаев она должна иметь средства программного повтора в коде приложения с ограничением количества попыток, равному, например 10 повторам. В любом случае, если транзакция зависит от содержимого базы данных, извлечённого в каких-либо предыдущих транзакциях SQL в рамках той же самой пользовательской транзакции, и некоторые параллельные транзакции изменили содержимое базы данных и по этой причине текущей транзакции не удастся обновить содержимое, то такая транзакция не должна иметь средства программного повтора, но управление должно быть возвращено пользователю для возможного перезапуска всей транзакции целиком. Об этом будет рассказано в статье «RVV Paper».

Новое соединение с базой данных для повтора транзакции SQL должно быть открыто в том случае, если соединение было разорвано из-за определённых проблем в сети.

Дополнительная литература, ссылки и загрузки

Дополнительная литература и ссылки

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. & O'Neil, P., "A Critique of ANSI SQL Isolation Levels", Technical Report MSR-TR-95-51, Microsoft Research, 1995.
Находится в свободном доступе на многих сайтах.

Delaney, K., “SQL Server Concurrency – Locking, Blocking and Row Versioning”, Simple Talk Publishing, July 2012

Melton, J. & Simon, A. R., "SQL:1999: Understanding Relational Language components", Morgan Kaufmann, 2002.

Data Management: Structured Query Language (SQL) Version 2, The Open Group, 1996. Доступно по адресу **http://www.opengroup.org/onlinepubs/9695959099/toc.pdf**

Избранные труды DBTechNet

Для получения дополнительной информации о технологиях управления параллелизмом в DB2, Oracle и SQL Server смотрите следующий документ по адресу:

http://www.dbtechnet.org/papers/SQL_ConcurrencyTechnologies.pdf

Более подробную информацию о транзакциях SQL как части пользовательской транзакции и применения проверки версий строк (RVV, также известна как «оптимистическая блокировка») в различных технологиях доступа к данным смотрите статью «RVV Paper» по адресу:

http://www.dbtechnet.org/papers/RVV_Paper.pdf

Virtual Database Laboratory Downloads

1. Файл OVA нашей виртуальной лаборатории баз данных «DebianDB»:

http://www.dbtechnet.org/download/DebianDBVM05.zip (3.9 GB, MySQL 5.1)

http://www.dbtechnet.org/download/DebianDBVM06.zip (4.8 GB, including MySQL 5.6, DB2 Express-C 9.7, Oracle XE 10.1, PostgreSQL 8.4, Pyrrho 4.8, and hands-on scripts)

2. «Краткое руководство» для доступа к продуктам СУБД, установленным в лаборатории:

http://www.dbtechnet.org/download/QuickStartToDebianDB.pdf

3. Скрипты для экспериментов из приложения 1, применимые к DB2, Oracle, MySQL и PostgreSQL:

http://www.dbtechnet.org/download/SQL_Transactions_Appendix1.zip

Приложение 1 Эксперименты с транзакциями SQL Server

Итак, добро пожаловать в «Обзорный тур в мир транзакций MySQL» с использованием ваших любимых продуктов СУБД, с целью поэкспериментировать и проверить себя на знание того, что было представлено в этой обучающей программе. Продукты СУБД могут вести себя по-разному в ходе транзакций, а некоторые моменты могут удивить вас и ваших клиентов – в том случае, если при разработке приложения вы ничего не будете знать об этих различиях. Если у вас есть время для ознакомления с большим количеством продуктов СУБД, то у вас будет и более широкий обзор сервисов транзакций, предоставляемыми продуктами СУБД на рынке.

Примечание: Скрипты, написанные с учётом особенностей продуктов СУБД, доступных в нашей лаборатории DebianDB, можно найти по адресу:

http://www.dbtechnet.org/download/SQL_Transactions_Appendix1.zip

В этом приложении мы представляем ряд наших экспериментов, выполненный на SQL Server Express 2012. Поскольку SQL Server работает только на платформах Windows и не доступен в DebianDB, то мы также представляем большинство результатов упражнений, чтобы вы могли сравнить их результаты со своими собственными экспериментами. Если вы хотите проверить наши результаты, вы можете загрузить SQL Server Express 2012 бесплатно с сайта Microsoft для вашей рабочей станции Windows (для Windows 7 или более поздней версии).

В следующих экспериментах мы используем программу SQL Server Management Studio (SSMS). Во-первых, мы создадим новую базу данных «TestDB» как это показано ниже и будем использовать установки по умолчанию для всех параметров конфигурации, а при помощи команды USE начнем использовать её в качестве текущей базы данных в нашей SQL-сессии.

```
-----------------------------------
CREATE DATABASE TestDB;
USE TestDB;
-----------------------------------
```

ЧАСТЬ 1. ЭКСПЕРИМЕНТЫ С ОДИНОЧНЫМИ ТРАНЗАКЦИЯМИ

По умолчанию, в SQL Server сессии работают в режиме AUTOCOMMIT (автоматической фиксации) и использования явных транзакций, мы можем построить транзакции нескольких SQL-команд. Тем не менее, весь сервер может быть настроен на использование и неявных транзакций. Но возможно использование неявных транзакций и только в одной сессии SQL, что можно настроить с помощью следующей команды SQL

```
SET IMPLICIT_TRANSACTIONS ON;
```

которая будет в силе до конца сессии, но при необходимости её можно будет выключить следующей командой:

```
SET IMPLICIT_TRANSACTIONS OFF;
```

Теперь начнем эксперименты с одиночными транзакциями. В самом начале мы также покажем некоторые важные результаты:

```
-----------------------------------------------------------------
-- Упражнение 1.1
-----------------------------------------------------------------
--  Autocommit mode
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), si SMALLINT);
Command(s) completed successfully.

INSERT INTO T (id, s) VALUES (1, 'first');
(1 row(s) affected)

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)

ROLLBACK; -- What happens?
Msg 3903, Level 16, State 1, Line 3
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)


BEGIN TRANSACTION;  -- An explicit transaction begins

INSERT INTO T (id, s) VALUES (2, 'second');

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           second                         NULL
(2 row(s) affected)

ROLLBACK;

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)
```

В транзакции сработала команда отката (ROLLBACK), но теперь мы вернулись в режим автоматической фиксации (AUTOCOMMIT)!

```
-------------------------------------------------------------------
-- Упражнение 1.2
-------------------------------------------------------------------
INSERT INTO T (id, s) VALUES (3, 'third');
(1 row(s) affected)

ROLLBACK;
Msg 3903, Level 16, State 1, Line 3
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
3           third                          NULL
(2 row(s) affected)

COMMIT;
Msg 3902, Level 16, State 1, Line 2
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.



-------------------------------------------------------------------
-- Упражнение 1.3
-------------------------------------------------------------------
BEGIN TRANSACTION;

DELETE FROM T WHERE id > 1;
(1 row(s) affected)

COMMIT;

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL

(1 row(s) affected)


-------------------------------------------------------------------
-- Упражнение 1.4
-- DDL stands for Data Definition Language. In SQL, the statements like
-- CREATE, ALTER and DROP are called DDL statements.
-- Now let's test use of DDL commands in a transaction!
-------------------------------------------------------------------
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO T (id, s) VALUES (2, 'will this be committed?');
CREATE TABLE T2 (id INT);  -- testing use of a DDL command in atransaction!
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;
GO  -- GO marks the end of a batch of SQL commands to be sent to the server
(1 row(s) affected)

(1 row(s) affected)
```

```
id
-----------
1
(1 row(s) affected)

SELECT * FROM T;    -- What has happened to T ?
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)

SELECT * FROM T2;   -- What has happened to T2 ?
Msg 208, Level 16, State 1, Line 2
Invalid object name 'T2'.



------------------------------------------------------------------
-- Упражнение 1.5a
------------------------------------------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;
------------------------------------------------------------------
-- Testing if an error would lead to automatic rollback in SQL Server.
--   @@ERROR is the SQLCode indicator in Transact-SQL, and
--   @@ROWCOUNT is the count indicator of the effected rows
------------------------------------------------------------------
INSERT INTO T (id, s) VALUES (2, 'The test starts by this');
(1 row(s) affected)

SELECT 1/0 AS dummy;   -- Division by zero should fail!
dummy
-----------
Msg 8134, Level 16, State 1, Line 1
Divide by zero error encountered.

SELECT @@ERROR AS 'sqlcode'
sqlcode
-----------
8134
(1 row(s) affected)

UPDATE T SET s = 'foo' WHERE id = 9999;  -- Updating a non-existing row
(0 row(s) affected)

SELECT @@ROWCOUNT AS 'Updated'
Updated
-----------
0
(1 row(s) affected)

DELETE FROM T WHERE id = 7777;   -- Deleting a non-existing row
(0 row(s) affected)

SELECT @@ROWCOUNT AS 'Deleted'
Deleted
-----------
```

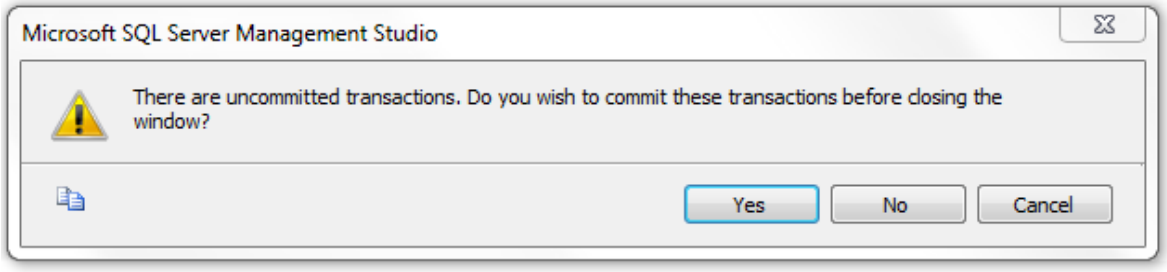
```
0
(1 row(s) affected)

COMMIT;

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           The test starts by this        NULL
(2 row(s) affected)

INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate')
INSERT INTO T (id, s) VALUES (3, 'How about inserting too long string
value?')
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
SELECT * FROM T;
COMMIT;
GO

Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK__T__3213E83FD0A494FC'. Cannot insert
duplicate key in object 'dbo.T'. The duplicate key value is (2).
The statement has been terminated.
Msg 8152, Level 16, State 14, Line 2
String or binary data would be truncated.
The statement has been terminated.
Msg 220, Level 16, State 1, Line 3
Arithmetic overflow error for data type smallint, value = 32769.
The statement has been terminated.
Msg 8152, Level 16, State 14, Line 4
String or binary data would be truncated.
The statement has been terminated.
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           The test starts by this        NULL
(2 row(s) affected)

BEGIN TRANSACTION;
SELECT * FROM T;
DELETE FROM T WHERE id > 1;
COMMIT;



-----------------------------------------------------------------
-- Упражнение 1.5b
-- This is special to SQL Server only!
-----------------------------------------------------------------
SET XACT_ABORT ON;  -- In this mode an error generates automatic rollback
SET IMPLICIT_TRANSACTIONS ON;

SELECT 1/0 AS dummy;   -- Division by zero

INSERT INTO T (id, s) VALUES (6, 'insert after arithm. error');
```

```
COMMIT;

SELECT @@TRANCOUNT AS 'do we have an transaction?'
GO

dummy
-----------
Msg 8134, Level 16, State 1, Line 3
Divide by zero error encountered.

SET XACT_ABORT OFF; -- In this mode an error does not generate automatic rollback

SELECT * FROM T;
id          s                               si
----------- ------------------------------- ------
1           first                           NULL
2           The test starts by this         NULL
(2 row(s) affected)

-- What happened to the transaction?


-----------------------------------------------------------------
-- Упражнение 1.6   Experimenting with Transaction Logic
-----------------------------------------------------------------
SET NOCOUNT ON;  -- Skipping the "n row(s) affected" messages
DROP TABLE Accounts;
SET IMPLICIT_TRANSACTIONS ON;

CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL
        CONSTRAINT unloanable_account CHECK (balance >= 0)
);

COMMIT;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

COMMIT;

-- Let's try the bank transfer
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         900
202         2100
```

```
ROLLBACK;

-- Let's test if the CHECK constraint actually works:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
Msg 547, Level 16, State 0, Line 2
The UPDATE statement conflicted with the CHECK constraint
"unloanable_account". The conflict occurred in database "TestDB", table
"dbo.Accounts", column 'balance'.
The statement has been terminated.

UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         4000

ROLLBACK;

-- Transaction logic using the IF structure of Transact-SQL
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
Msg 547, Level 16, State 0, Line 4
The UPDATE statement conflicted with the CHECK constraint
"unloanable_account". The conflict occurred in database "TestDB", table
"dbo.Accounts", column 'balance'.
The statement has been terminated.

IF @@error <> 0 OR @@rowcount = 0
   ROLLBACK
ELSE BEGIN
   UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
   IF @@error <> 0 OR @@rowcount = 0
      ROLLBACK
   ELSE
      COMMIT;
   END;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000
COMMIT;

-- How about using a non-existent bank account?
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;

SELECT * FROM Accounts ;
```

```
acctID      balance
----------- -----------
101         500
202         2000
ROLLBACK;

-- Fixing the case using the IF structure of Transact-SQL
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
IF @@error <> 0 OR @@rowcount = 0
   ROLLBACK
ELSE BEGIN
   UPDATE Accounts SET balance = balance + 500 WHERE acctID = 707;
   IF @@error <> 0 OR @@rowcount = 0
      ROLLBACK
   ELSE
      COMMIT;
   END;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000


-----------------------------------------------------------------------
-- Упражнение 1.7   Testing the database recovery
-----------------------------------------------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;

BEGIN TRANSACTION;
INSERT INTO T (id, s) VALUES (9, 'What happens if ..');

SELECT * FROM T;
id          s                             si
----------- ----------------------------- ------
1           first                         NULL
9           What happens if ..            NULL
```

При **завершении работы SQL Server Management Studio** мы увидим следующее сообщение



и для достижения целей нашего эксперимента мы выберем «No».

При **повторном запуске Management Studio** и подключении к нашей TestDB мы можем изучить, что произошло с нашей последней незавершенной транзакцией с помощью простого перечисления содержимого таблицы T.

```
SET NOCOUNT ON;
SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
```

ЧАСТЬ 2. ЭКСПЕРИМЕНТЫ С ПАРАЛЛЕЛЬНЫМИ ТРАНЗАКЦИЯМИ

Для эксперимента с параллельными транзакциями мы откроем два окна с параллельными запросами SQL и сессиями «Клиент А» и «Клиент В», имеющих доступ к той же базе данных TestDB. Для обеих сессий мы выбираем перечисление результатов в текстовом режиме с помощью следующих выборов меню:

Query > Results To > Results to Text

и в обеих сессиях установим использование неявных транзакций

```
SET IMPLICIT_TRANSACTIONS ON;
```

Для максимального удобства отображения окон параллельных запросов SQL в Management Studio, например, чтобы расположить одно окно рядом с другим вертикально, надо нажатием правой кнопки мыши либо в самом окне, либо на названии окна SQLQuery вызвать контекстное меню и выбрать там «New Vertical Tab Group» (см. рисунок А1.1).

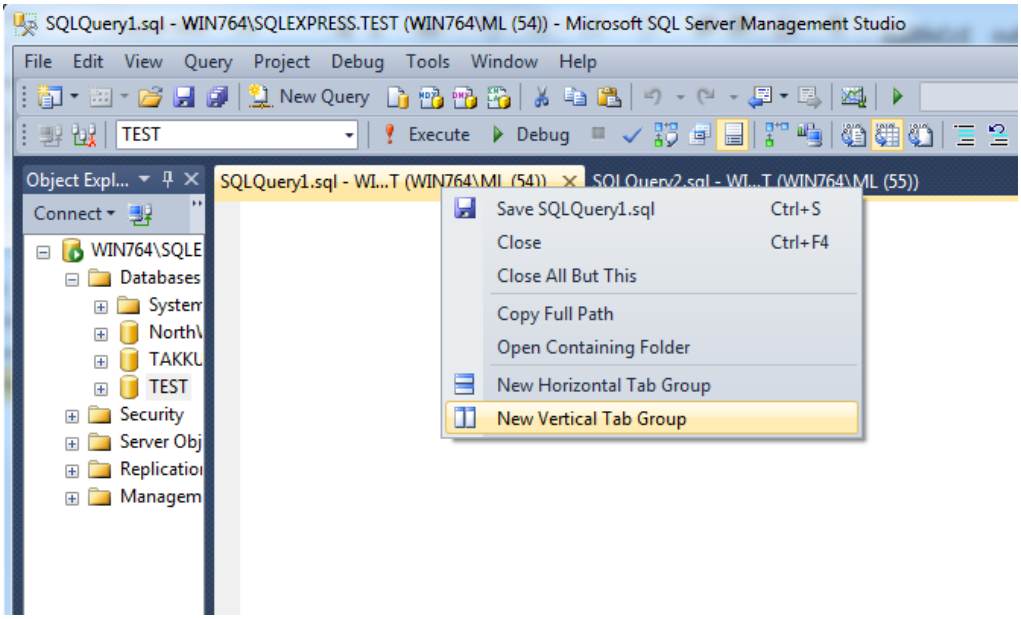


Рисунок А1.1 Расположение двух окон SQL-запросов рядом друг с другом

```
-----------------------------------------------------------------
-- Упражнение 2.1   "Lost update problem simulation"
-----------------------------------------------------------------
-- 0. To start with fresh contents we enter following commands on a session
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;
```

Проблема потерянных обновлений возникает, когда некоторые вставленные или обновлённые первой транзакцией строки будут обновлены или удалены какими-либо другими параллельными транзакции до завершения первой транзакции. Это возможно в файловых решениях «NoSQL», но современные СУБД не допустят возникновения этого

явления. Тем не менее, после фиксации первой транзакции, любая конкурирующая транзакция может перезаписать строки зафиксированных транзакций. Далее мы смоделируем сценарий потери обновления с помощью уровня изолированности READ COMMITTED, который не держит S-блокировки. Сначала клиент приложения считывает значения баланса, снимая S-блокировки.

```
-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

acctID      balance
----------- -----------
101         1000

-- 2. Client B starts
SET NOCOUNT ON;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
```

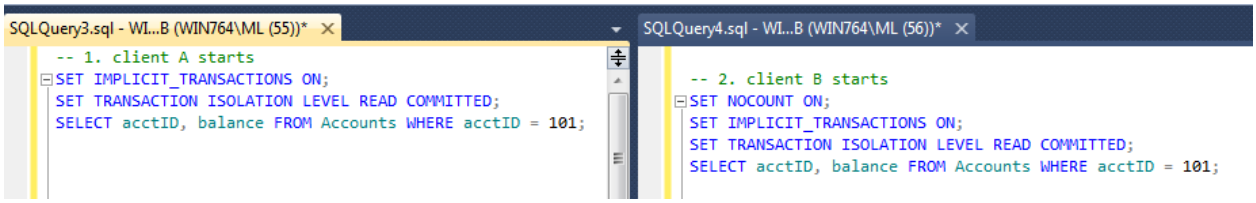
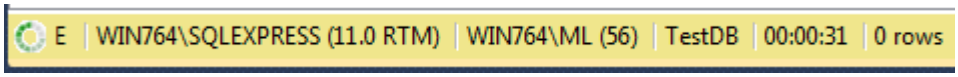


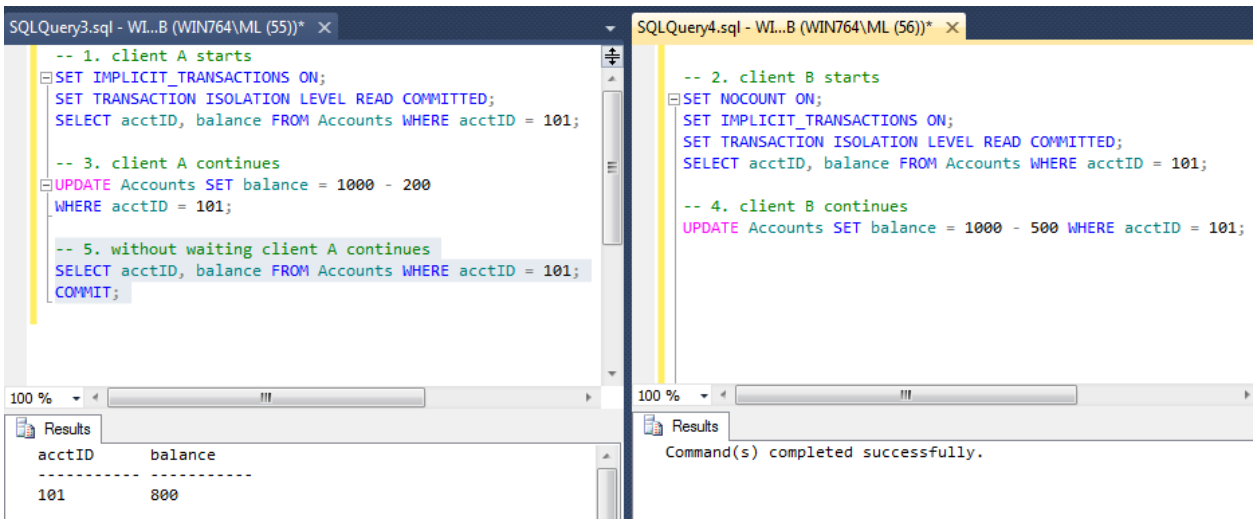
Рисунок А1.2 Конкурирующие SQL-сессии в своих окнах

```
-- 3. Client A continues
UPDATE Accounts SET balance = 1000 - 200 WHERE acctID = 101;

-- 4. Client B continues
UPDATE Accounts SET balance = 1000 - 500 WHERE acctID = 101;
```

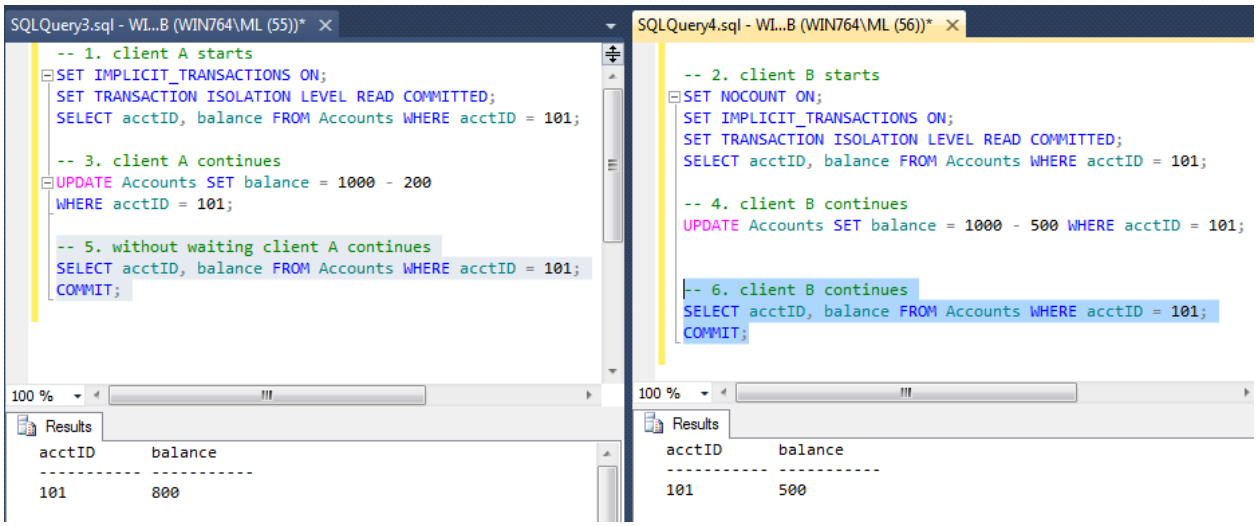


... в ожидании A ...

```
-- 5. Client A continues
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;
```




```
-- 6. Client B continues
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;
```

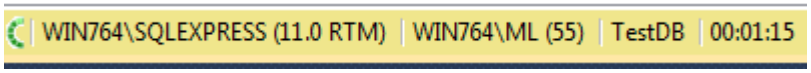


Таким образом, конечный результат является ошибочным!

> **Примечание**: В этом эксперименте у нас не было реальной проблемы «потерянного обновления», но после того, как A фиксирует свою транзакцию, B может продолжить, в результате чего переписывает обновление, выполненное A. Мы называем эту проблему надежности «слепой перезаписью» (Blind Overwriting) или «записью «грязных» данных» (Dirty Write). Это может быть устранено, если команды UPDATE используют чувствительные обновления вроде:
>
> ```
> SET balance = balance - 500
> ```

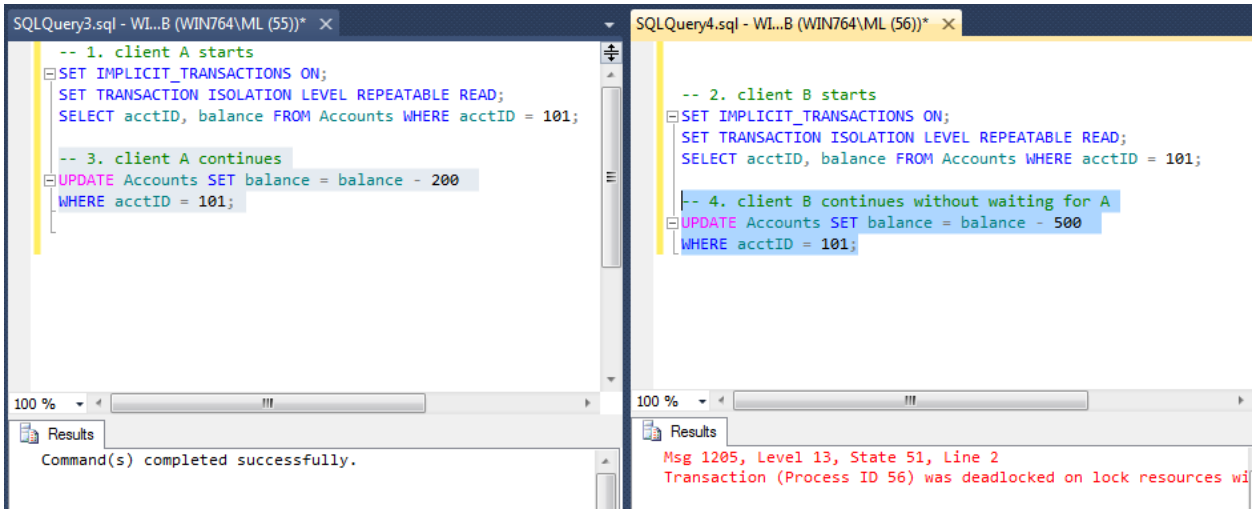
```
-----------------------------------------------------------------
-- Упражнение 2.2   "Lost Update Problem" fixed by locks
-----------------------------------------------------------------
--      Competition on a single resource
--      using SELECT .. UPDATE scenarios both client A and B
--      tries to withdraw amounts from the same account.
--
```

```
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- 3. client A continues
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 101;
```



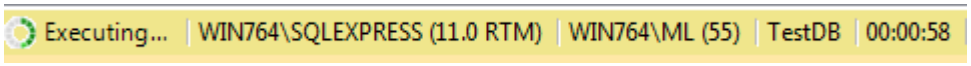
... в ожидании B ...

```
-- 4. Client B continues without waiting for A
UPDATE Accounts SET balance = balance - 500
WHERE acctID = 101;
```



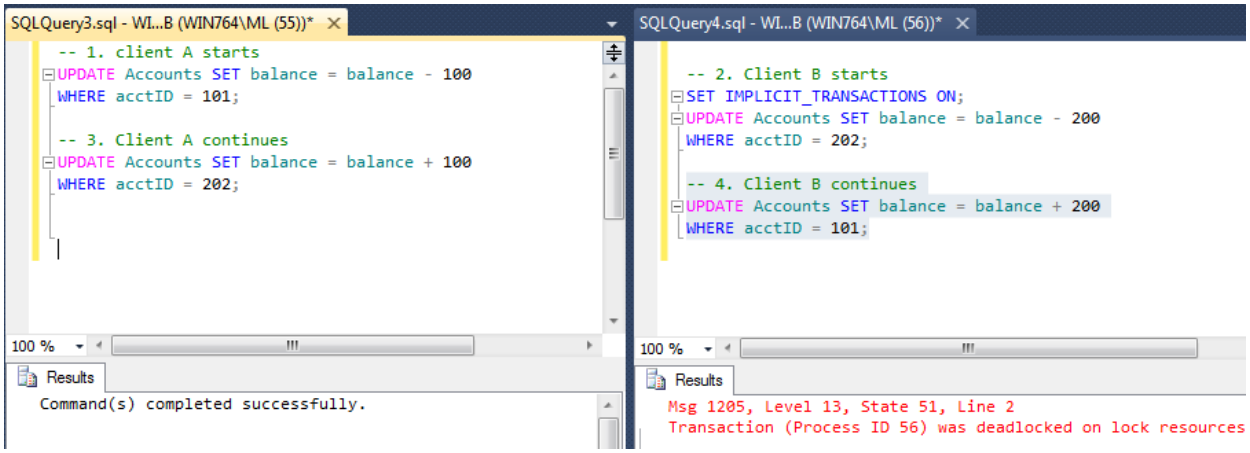

```
-- 5. The client which survived will commit
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctID      balance
----------- -----------
101         800

COMMIT;
```

```
-----------------------------------------------------------------
-- Упражнение 2.3  Competition on two resources in different order
--                 using UPDATE-UPDATE scenarios
-----------------------------------------------------------------
-- Client A transfers 100 euros from account 101 to 202
-- Client B transfers 200 euros from account 202 to 101
--
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;

-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 202;

-- 3. Client A continues
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 202;

COMMIT;
```



... в ожидании B ...

```
-- 4. Client B continues
UPDATE Accounts SET balance = balance + 200
WHERE acctID = 101;
```




```
-- 5. Client A continues if it can ...
COMMIT;
```

В упражнениях 2.4 - 2.7 мы будем экспериментировать с **аномалиями параллелизма**, то есть рисками надежности данных, известные по стандарту ISO SQL. Сможем ли мы идентифицировать их? И как мы сможем устранить аномалии?

```
-----------------------------------------------------------------
-- Упражнение 2.4   Dirty Read?
-----------------------------------------------------------------
--
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;

UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 202;


-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         900
202         2100
COMMIT;


-- 3. Client A continues
ROLLBACK;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

COMMIT;
```

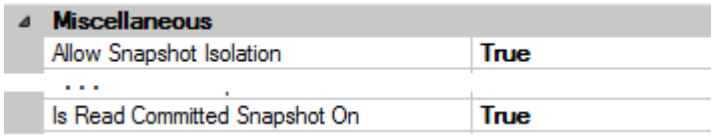
```
-----------------------------------------------------------------
-- Упражнение 2.5   Non-repeatable Read?
-----------------------------------------------------------------
-- In non-repeatable read anomaly some rows read in the current transaction
-- may not appear in the resultset if the read operation would be repeated
-- before end of the transaction.

-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

--  Listing accounts having balance > 500 euros
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
101         1000
202         2000


-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202;
COMMIT;


-- 3. Client A continues
-- Can we still see the same accounts as in step 1?
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
202         2500

COMMIT;
```

```
-----------------------------------------------------------------
-- Упражнение 2.6   Insert Phantom?
-----------------------------------------------------------------
-- Insert phantoms are rows inserted by concurrent transactions and
-- which the current might see before the end of the transaction.
--
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- Accounts having balance > 1000 euros
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
101         1000
202         2000


-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO Accounts (acctID,balance) VALUES (303,3000);
COMMIT;


-- 3. Client A continues
-- Let's see the results
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
202         2000
303         3000

COMMIT;
```

Вопрос

- Как мы можем предотвратить появление фантомов?

```
------------------------------------------------------------
-- SNAPSHOT STUDIES
------------------------------------------------------------
-- The database needs to be configured to support SNAPSHOT isolation.
-- For this we create a new database

CREATE DATABASE SnapsDB;

-- to be configured to support snapshots as follows
```

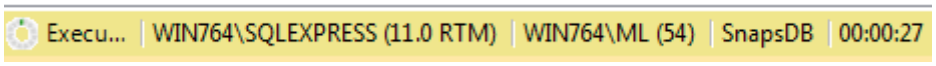



```
-- Then both client A and B are switched to use SnapsDB as follows:

USE SnapsDB;


--------------------------------------------------------------------
-- Упражнение 2.7   A Snapshot study with different kinds of Phantoms
--------------------------------------------------------------------
USE SnapsDB;

-- 0. Setup the test: recreate the table T with five rows
DROP TABLE T;
GO

SET IMPLICIT_TRANSACTIONS ON;
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), i SMALLINT);
INSERT INTO T (id, s, i) VALUES (1, 'first',  1);
INSERT INTO T (id, s, i) VALUES (2, 'second', 2);
INSERT INTO T (id, s, i) VALUES (3, 'third', 1);
INSERT INTO T (id, s, i) VALUES (4, 'forth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);
COMMIT;

-- 1. Client A starts
USE SnapsDB;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL SNAPSHOT ;

SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           first                          1
3           third                          1
5           to be or not to be             1
```

```
-- 2. Client B starts
USE SnapsDB;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

INSERT INTO T (id, s, i) VALUES (6, 'Insert Phantom', 1);
UPDATE T SET s = 'Update Phantom', i = 1 WHERE id = 2;
DELETE FROM T WHERE id = 5;

SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           first                          1
2           Update Phantom                 1
3           third                          1
4           forth                          2
6           Insert Phantom                 1


-- 3. Client A continues
-- Let's repeat the query and try some updates
SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           first                          1
3           third                          1
5           to be or not to be             1

INSERT INTO T (id, s, i) VALUES (7, 'inserted by A', 1);
UPDATE T SET s = 'update by A inside snapshot' WHERE id = 3;
UPDATE T SET s = 'update by A outside snapshot' WHERE id = 4;
UPDATE T SET s = 'update by A after B' WHERE id = 1;

SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           update by A after B            1
3           update by A inside snapshot    1
5           to be or not to be             1
7           inserted by A                  1
```

```
-- 3.5 Client C in a new session starts and executes a query
USE SnapsDB;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           update by A after B            1
2           Update Phantom                 1
3           update by A inside snapshot    1
4           update by A outside snapshot   2
6           Insert Phantom                 1
7           inserted by A                  1
(6 row(s) affected)
```

```
-- 4. Client B continues
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           first                          1
2           Update Phantom                 1
3           third                          1
4           forth                          2
6           Insert Phantom                 1


-- 5. Client A continues
SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           update by A after B            1
3           update by A inside snapshot    1
5           to be or not to be             1
7           inserted by A                  1

UPDATE T SET s = 'update after delete?' WHERE id = 5;
```



... в ожидании B ...

```
-- 6. Client B continues without waiting for A
COMMIT;

-- 7. Client A continues
Msg 3960, Level 16, State 2, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot use
snapshot isolation to access table 'dbo.T' directly or indirectly in database
'SnapsDB' to update, delete, or insert the row that has been modified or
deleted by another transaction. Retry the transaction or change the isolation
level for the update/delete statement.

-- 8. Client B continues
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           first                          1
2           Update Phantom                 1
3           third                          1
4           forth                          2
6           Insert Phantom                 1

(5 row(s) affected)
```

Вопрос

- Объясните ход эксперимента. Объясните различные результаты шагов 3, 5 и 4.

--

Более сложные темы по транзакциям SQL Server смотрите:
Kalen Delaney (2012), “SQL Server Concurrency, Locking, Blocking and Row Versioning”
ISBN 978-1-906434-90-8

Приложение 2 Транзакции в программирования на языке Java

Мы можем экспериментировать с транзакциями SQL как доступными на диалекте SQL и в сервисах продукта СУБД, которые будут использоваться при помощи интерактивного SQL-клиента, именуемого SQL Editors. Однако, для приложений доступ к данным запрограммирован с использованием определённых API. Чтобы дать короткий пример написания транзакций SQL с использованием определённых API, мы представляем пример Банковского перевода, реализованный на Java с использованием JDBC (англ. Java DataBase Connectivity - соединение с базами данных на Java) в качестве API доступа к данным. Мы предполагаем, что читатель этого приложения уже знаком с программированием на Java и JDBC API, а рисунок А2.1 служит только в качестве визуального краткого обзора, помогающего запомнить главные объекты, методы и их взаимодействие.

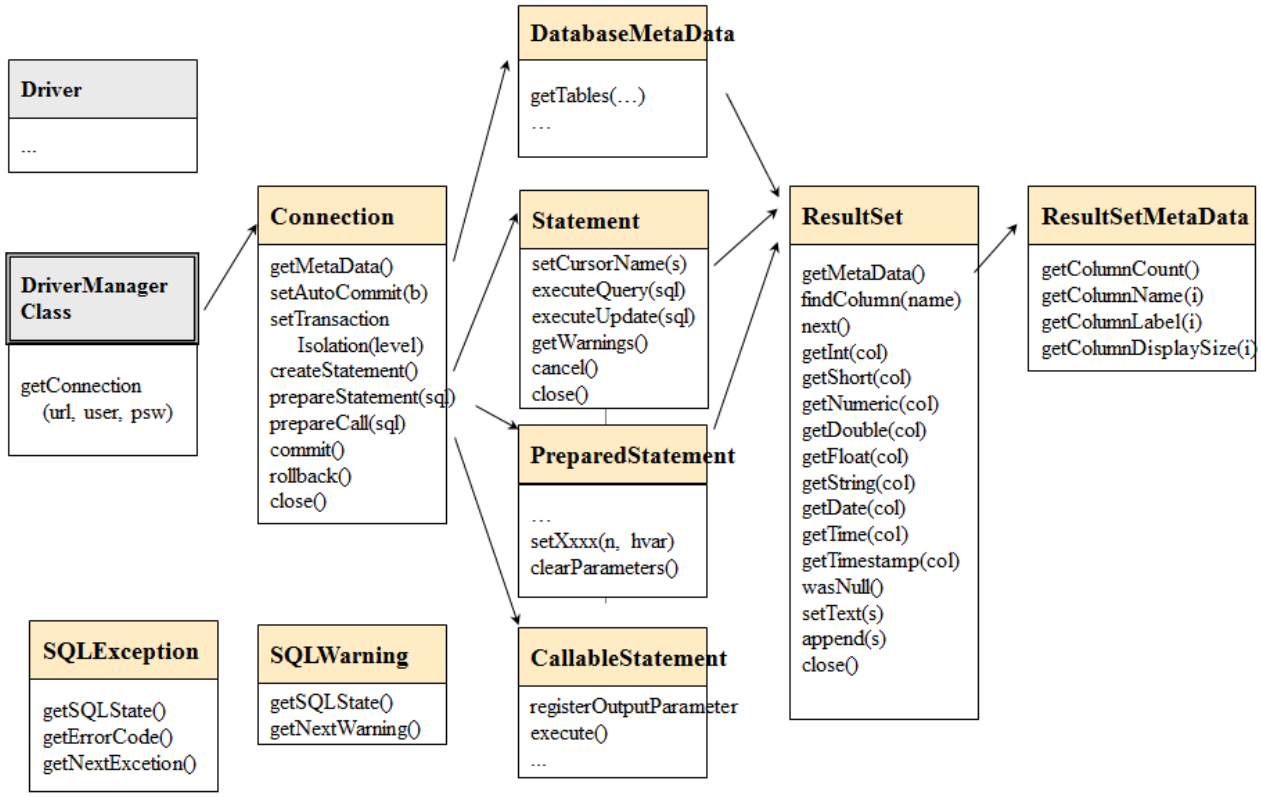


Рисунок А2.1 Упрощённый обзор JDBC API

Пример кода Java: банковский перевод

Программа BankTransfer выполняет перевод суммы 100 евро с одного банковского счета (fromAcct) на другой (toAcct). Название драйвера, URL базы данных, имени пользователя, пароля пользователя, параметров fromAcct и toAcct считываются из параметров командной строки программы.

Для проведения теста пользователь должен открыть два окна командной строки в случае платформы Windows или два окна терминала в случае платформы Linux и управлять программой одновременно в этих окнах так, чтобы fromAcct первого был toAcct другого и наоборот, как это видно в скриптах ниже.

After updating the fromAcct the program waits for the ENTER key to continue so that test runs get synchronized and we can test the concurrency conflict. The source program demonstrates also the ReTryer data access pattern.

После обновления fromAcct программа ждет нажатия ENTER с тем, чтобы продолжение испытания было синхронизировано, таким образом мы сможем проверить конфликт параллелизма. Исходная программа демонстрирует также **образец доступа к данным ReTryer**.

Перечень A2.1 Пример кода банковского перевода на Java с использованием JDBC

```
/*  DBTechNet Concurrency Lab 15.5.2008 Martti Laiho

   BankTransfer.java

Save the java program as BankTransfer.java and compile as follows

javac BankTransfer.java

See BankTransferScript.txt for the test scripts applied to SQL Server, Oracle and DB2

Updates:
2.0 2008-05-26 ML preventing rollback by application after SQL Server deadlock
2.1 2012-09-24 ML restructured for presenting the Retry Wrapper block
2.2 2012-11-04 ML exception on non-existing accounts
**********************************************************/
import java.io.*;
import java.sql.*;
public class BankTransfer {
   public static String moreRetries = "N";

   public static void main (String args[]) throws Exception
   {
      System.out.println("BankTransfer version 2.2");

      if (args.length != 6) {
         System.out.println("Usage:" +
         "BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%");
         System.exit(-1);
      }
      java.sql.Connection conn = null;
     // String moreRetries = "N";
      boolean sqlServer = false;
      int counter = 0;
      String URL = args[1];
      String user = args[2];
      String password = args[3];
      String errMsg = "";
      int amount = 100;
      int fromAcct = Integer.parseInt(args[4]);
      int toAcct = Integer.parseInt(args[5]);
```

```
   // SQL Server's explicit transactions will require special treatment
   if (URL.substring(5,14).equals("sqlserver")) {
     sqlServer = true;
   }
         // register the JDBC driver and open connection
   try {
      Class.forName(args[0]);
      conn = java.sql.DriverManager.getConnection(URL,user,password);
   }
   catch (SQLException ex) {
        System.out.println("URL: " + URL);
        System.out.println("** Connection failure: " + ex.getMessage() +
                           "\n SQLSTATE: " + ex.getSQLState() +
                           " SQLcode: " + ex.getErrorCode());
      System.exit(-1);
   }
```

```
   do
   {

   // Retry wrapper block of TransaferTransaction
      if (counter++ > 0) {
         System.out.println("retry #" + counter);
         if (sqlServer) {
            conn.close();
            System.out.println("Connection closed");
            conn = java.sql.DriverManager.getConnection(URL,user,password);
            conn.setAutoCommit(true);
         }
      }
      TransferTransaction (conn,
                           fromAcct, toAcct, amount,
                           sqlServer, errMsg //,moreRetries
                           );
      System.out.println("moreRetries="+moreRetries);
      if (moreRetries.equals("Y")) {
          long pause = (long) (Math.random () * 1000); // max 1 sec.
          System.out.println("Waiting for "+pause+ " mseconds"); // just for testing
          Thread.sleep(pause);
      }
   } while (moreRetries.equals("Y") && counter < 10);  // max 10 retries
   // end of the Retry wrapper block
```

```
   conn.close();
   System.out.println("\n End of Program. ");
}

static void TransferTransaction (Connection conn,
      int fromAcct, int toAcct, int amount,
      boolean sqlServer,
      String errMsg //, String moreRetries
      )
      throws Exception {
```

```
String SQLState = "*****";
try {
    conn.setAutoCommit(false);   // transaction begins
    conn.setTransactionIsolation(
        Connection.TRANSACTION_SERIALIZABLE);
    errMsg = "";
    moreRetries = "N";

    // a parameterized SQL command
    PreparedStatement pstmt1 = conn.prepareStatement(
      "UPDATE Accounts SET balance = balance + ? WHERE acctID = ?");
    // setting the parameter values
    pstmt1.setInt(1, -amount);  // how much money to withdraw
    pstmt1.setInt(2, fromAcct); // from which account
    int count1 = pstmt1.executeUpdate();
    if (count1 != 1) throw new Exception ("Account "+fromAcct + " is missing!");
    // --- Interactive pause for concurrency testing -----
    // In the following we arrange the transaction to wait
    // until the user presses ENTER key so that another client
    // can proceed with a conflicting transaction.
    // This is just for concurrency testing, so don't apply this
    // user interaction in real applications!!!
      System.out.print("\nPress ENTER to continue ...");
      BufferedReader reader = new BufferedReader(
              new InputStreamReader(System.in));
      String s = reader.readLine();
    // --- end of waiting ------------------------------------
    pstmt1.setInt(1, amount);  // how much money to add
    pstmt1.setInt(2, toAcct);  // to which account
    int count2 = pstmt1.executeUpdate();
    if (count2 != 1) throw new Exception  ("Account "+toAcct + " is missing!");
    System.out.print("committing ..");
    conn.commit();  // end of transaction
    pstmt1.close();
 }
 catch (SQLException ex) {
    try {
       errMsg = "\nSQLException:";
       while (ex != null) {
            SQLState = ex.getSQLState();
            // is it a concurrency conflict?
            if ((SQLState.equals("40001")   // Solid, DB2, SQL Server,...
              || SQLState.equals("61000")   // Oracle ORA-00060: deadlock detected
              || SQLState.equals("72000"))) // Oracle ORA-08177: can't serialize access
                 moreRetries = "Y";
            errMsg = errMsg + "SQLState: "+ SQLState;
            errMsg = errMsg + ", Message:  "+ ex.getMessage();
            errMsg = errMsg + ", Vendor:   "+ ex.getErrorCode() + "\n";
            ex = ex.getNextException();
       }
       // SQL Server does not allow rollback after deadlock !
```

```
            if (sqlServer == false) {
               conn.rollback();  // explicit rollback needed for Oracle
                        // and the extra rollback does not harm DB2
            }
            // println for testing purposes
            System.out.println("** Database error: " + errMsg);
         }
         catch (Exception e) { // In case of possible problems in SQLException handling
            System.out.println("SQLException handling error: " + e);
            conn.rollback(); // Current transaction is rolled back
            ;  // This is reserved for potential exception handling
         }
      } // SQLException
      catch (Exception e) {
         System.out.println("Some java error: " + e);
         conn.rollback(); // Current transaction is rolled back also in this case
         ;  // This is reserved for potential other exception handling
      } // other exceptions
   }
}
```

Скрипты в перечне A2.2 могут использоваться для проверки программы на рабочей станции Windows в двух параллельных окнах командной строки. Скрипты предполагают, что продуктом СУБД является SQL Server Express, база данных имеет название «TestDB», а драйвер JDBC хранится в подкаталоге «jdbc-drivers» текущего каталога программы.

Перечень A2.2 Скрипты для экспериментов с банковским переводом на платформе Windows

```
rem Script for the first window:
set CLASSPATH=.;jdbc-drivers\sqljdbc4.jar
set driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
set
URL="jdbc:sqlserver://localhost;instanceName=SQLEXPRESS;databaseName=TestDB"
set user="user1"
set password="sql"
set fromAcct=101
set toAcct=202
java BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%


rem Script for the second window:
set CLASSPATH=.;jdbc-drivers\sqljdbc4.jar
set driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
set
URL="jdbc:sqlserver://localhost;instanceName=SQLEXPRESS;databaseName=TestDB"
set user="user1"
set password="sql"
set fromAcct=202
set toAcct=101
java BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%
```

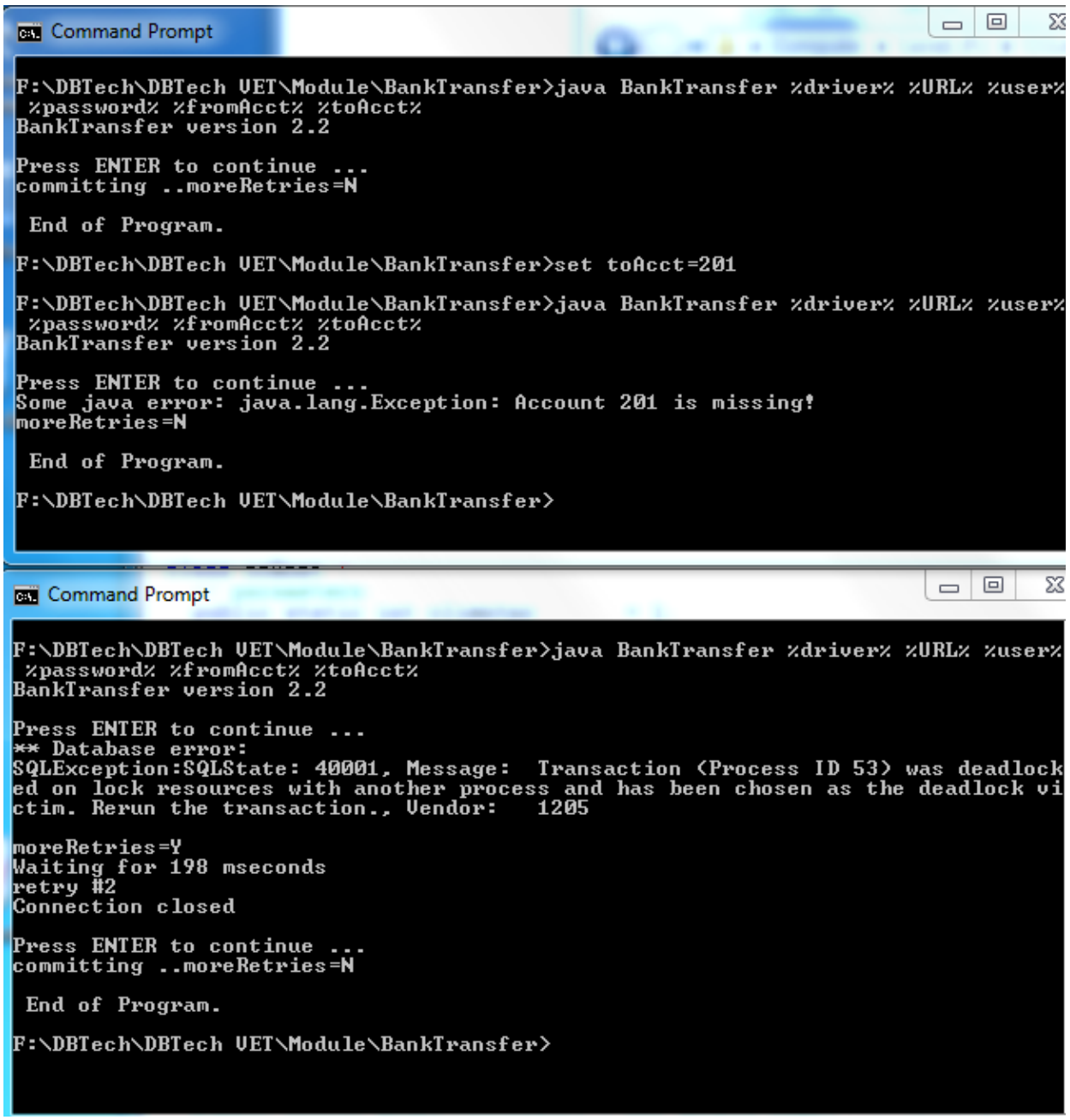


Рисунок А2.1 Образец теста банковского перевода на Windows

Скрипты для других продуктов СУБД и платформы Linux могут быть легко получены из скриптов, показанных в перечне А2.2.

Перечень А2.3 Скрипты для экспериментов с банковским переводом BankTransfer с использованием MySQL DebianDB на платформе Linux, где код BankTransfer.java был помещён и скомпилирован в каталоге Transactions пользователя студента, а драйверы JDBC были установлены в каталоге /opt/jdbc-drivers как показано ниже

Скрипты для MySQL на Linux:

```
# Creating directory /opt/jdbc-drivers for JDBC drivers
cd /opt
mkdir jdbc-drivers
chmod +r+r+r jdbc-drivers
# copying the MySQL jdbc driver to /opt/jdb-drivers
cd /opt/jdbc-drivers
cp $HOME/mysql-connector-java-5.1.23-bin.jar
# allow read access to the driver to everyone
chmod +r+r+r mysql-connector-java-5.1.23-bin.jar


#******* MySQL/InnoDB **************************************************

# First window:
export CLASSPATH=/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/testdb
export user=user1
export password=sql
export fromAcct=101
export toAcct=202
java -classpath .:$CLASSPATH BankTransfer $driver $URL $user $password
$fromAcct $toAcct


# Second window:
export CLASSPATH=/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/testdb
export user=user1
export password=sql
export fromAcct=202
export toAcct=101
java -classpath .:$CLASSPATH BankTransfer $driver $URL $user $password
$fromAcct $toAcct
#*********************************************************************
```

Приложение 3 Транзакции и восстановление базы данных

Рисунок А3.1 представляет "большую картину" того, что находится внутри обработки транзакций для типичного сервера базы данных. Учебные слайды «Basics of SQL Transactions» доступных по адресу:

http://www.dbtechnet.org/papers/BasicsOfSqlTransactions.pdf

представляют более подробно архитектуру некоторых основных продуктов СУБД, в том числе управление файлами баз данных и журналов транзакций (например, снимки журнала транзакций SQL); управление буферными пулами (кэш данных) в памяти для уменьшения количества операций ввода/вывода диска с целью повышения производительности; способы обсуживания транзакций SQL; надежность операций фиксации (COMMIT) и реализации операций отката (ROLLBACK).

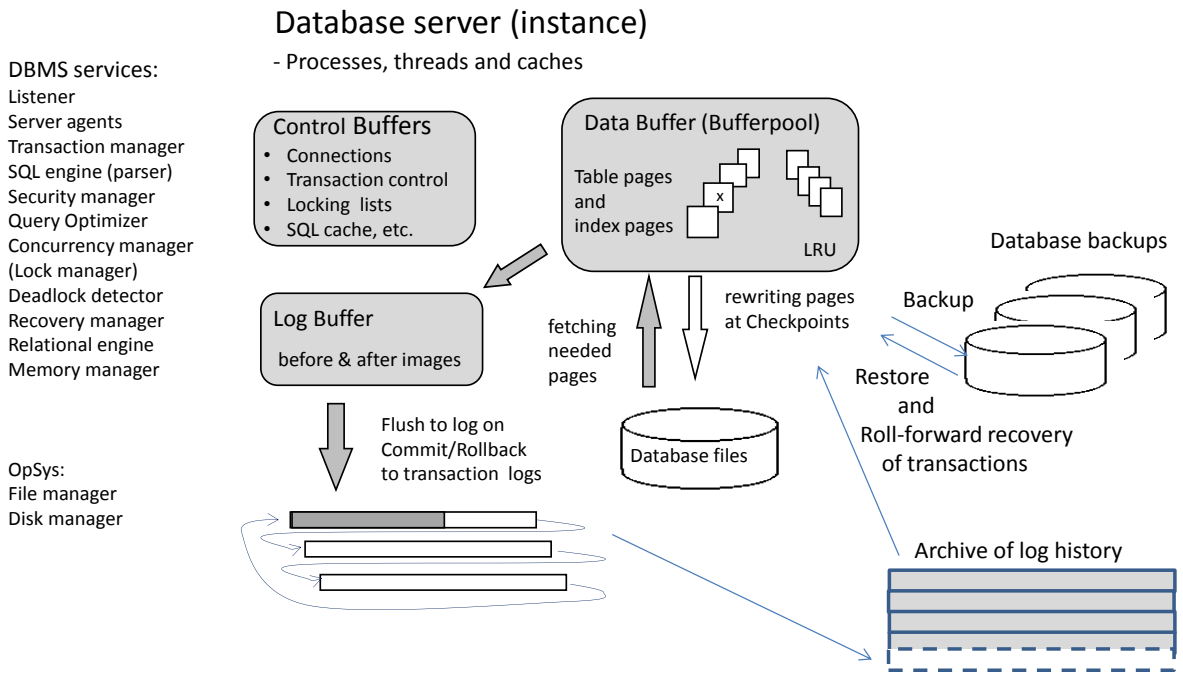


Рисунок А3.1 Обобщённый обзор сервера баз данных

Далее мы опишем, как СУБД может восстановить базу данных вплоть до последней зафиксированной транзакции в случае отключения электропитания и аварийного отказа программного обеспечения сервера. В случае аппаратных сбоев база данных может быть восстановлена с помощью резервной копии базы данных и сохраненных записей в истории журнала транзакций, начиная с того момента, когда была выполнена резервная копия базы данных.

(Смотрите также: **http://www.dbtechnet.org/papers/RdbmsRecovery.ppt**)

В начале транзакции SQL сервер базы данных присваивает ей уникальный идентификационный номер транзакции, а каждое действие в транзакции записывается в файл журнала транзакций. Для каждой обработанной строки запись журнала содержит идентификатор транзакции и исходное содержимое строки («изображение до») и содержимое строки после операции («изображение после»). В случае команд INSERT будут пусты части «изображения до», а в случае команды DELETE будут пусты части «изображения после». Кроме того, для операций фиксации (COMMIT) и отката

(ROLLBACK), будут произведены соответствующие записи в журнале, в конечном итоге все записи транзакции будут записаны в файл журнала транзакций на диске. С момента операции фиксации (COMMIT) управление будет возвращено клиенту только после записи фиксации на диск.

Время от времени сервер баз данных выполняет операцию CHECKPOINT (контрольная точка), во время которой обслуживание клиентов приостанавливается, все записи журнала транзакций из кэша журнала будет записываться в файл журнала транзакций и все обновленные страницы данных (отмеченные посредством «Dirty Bit» - англ. «грязный» бит) в кэше данных (буферных пулах) будут записаны на свои прежние местах в файлах данных в базе данных, а "грязные биты" этих страниц будет стёрты из кэша данных. Группа записей CHECKPOINT, в том числе список идентификационных номеров транзакций, выполняющихся в данный момент, будут записаны в журнале транзакций. После этого обслуживание клиентов возобновляется.

Примечание: При управляемом отключении сервера (managed shutdown) не должно быть никаких активных сеансов SQL, так что в качестве последней операции журнала транзакций сервер записывает пустую контрольную точку, указывающую на «чистое» отключение сервера.

Рисунок А3.2 показывает историю в журнале транзакций до остановки сервера, например, из-за перебоя в питании. Все файлы на диске могут быть прочитаны, но содержание в в кэше данных (буферных пулах) будет потеряно. Ситуацию называют «мягкий сбой системы» (аварийный отказ программного обеспечения).

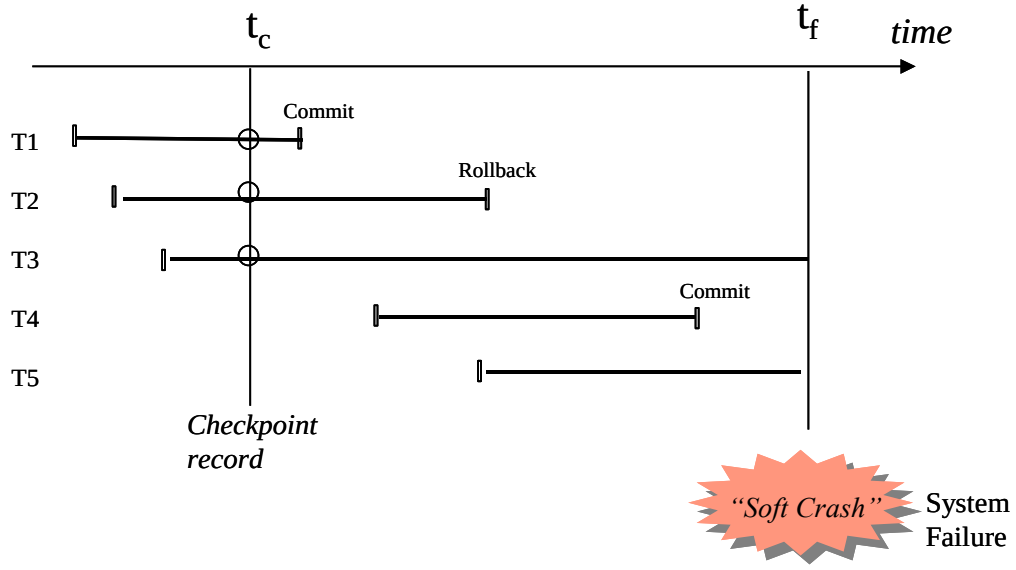


Рисунок А3.2 История в журнале транзакций до аварийного отказа программного обеспечения

После перезапуска сервер будет искать последнюю запись контрольной точки в журнале транзакций. Если записи контрольной точки нет, то это указывает на «чистое» отключение сервера, а сам сервер готов к обслуживанию клиентов. В противном случае сервер запускает процедуру отката с восстановлением следующим образом: во-первых, все номера транзакций, указанных в записи контрольной точки будут скопированы в список отмены транзакций (UNDO), а список идентификационных номеров транзакций, переданных в базу данных (REDO), будет стёрт. Сервер сканирует журнал транзакций от контрольной точки до конца журнала с перемещением идентификаторов всех новых

транзакций в список отмены (UNDO) и перемещением идентификаторов зафиксированных транзакций из списка отмены (UNDO) в список повтора (REDO). После этого сервер переходит назад в журнал и записывает в базу данных «изображения до» всех строк транзакций, перечисленным в списке отмены (UNDO), а затем переходит вперед от записи контрольной точки и записывает «изображения после» всех строк транзакций, перечисленным в REDO списке. После этого база данных была восстановлена в уровне последней зафиксированной транзакции до аварийного отказа программного обеспечения, и сервер может начать обслуживать клиентов (см. рисунок А3.3).

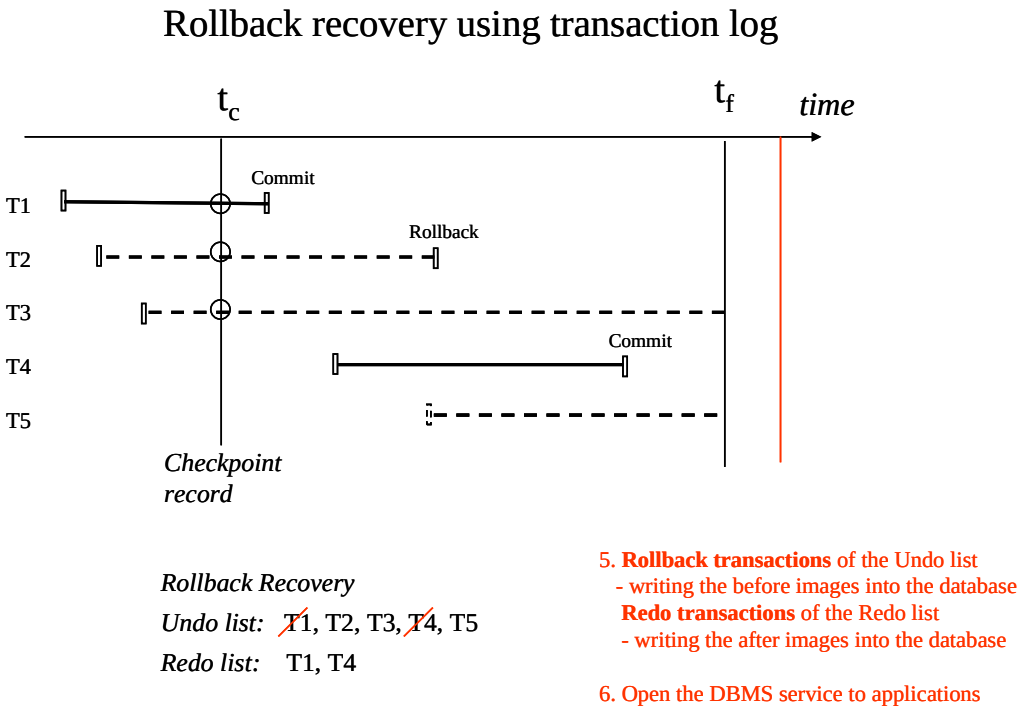


Рисунок А3.3 Процедура отката с восстановлением базы данных

Примечание: Наряду с упрощенной процедурой восстановления, описанной выше, большинство современных СУБД продукты используют протокол ARIES, в котором между контрольными точками, когда нет данных для загрузки в базу данных, поток «отложенной записи» синхронизирует «грязные» страницы из кэша данных обратно в файлы данных .Синхронизированные страницы, отмеченные знаками LSN может быть исключена из отката и восстановления, что сделает процедуру отката и восстановления быстрее.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- phantom read problem - проблема фантомного чтения;
- PL/SQL (Procedural Language/Structured Query Language) - язык программирования, процедурное расширение языка SQL;
- Read Committed - чтение фиксированных данных;
- Read Only transaction - транзакция, обозначенная только для чтения;
- Read Uncommitted - чтение незафиксированных данных;
- Repeatable Read - повторяемость чтения;
- retry wrapper - средства программного повтора;
- retry wrapper block - блок средств программного повтора;
- ROLLBACK;
- round trip - полный цикл;
- SCN - (англ. «System Change Number» - системный номер изменения);
- sensitive update - уязвимое обновление;
- serializable - упорядочиваемость;
- S-lock - совмещаемая блокировка;
- snapshot - снимок;
- soft crash - мягкий сбой системы (аварийный отказ программного обеспечения);
- SQL client - клиент SQL;
- SQL command - команда SQL;
- SQL Server - сервер SQL;
- SQL session - SQL-сессия;
- SQL statement - оператор SQL;
- SQLCODE (возвращает числовой код самого последнего исключения);
- SQLException (предоставляет информацию об ошибке доступа к базе данных или других ошибках);
- SQLSTATE;
- SQLWarning (предоставляет информацию о предупреждениях доступа к базе данных);
- stored procedures - записанные процедуры;
- stored routine - записанная подпрограмма;
- System Change Number - системный номер изменения;
- transaction id - идентификационный номер транзакции;
- transaction log - журнал транзакций;
- unit of recovery - единица восстановления;
- UNLOCK TABLE;
- user transaction - пользовательская транзакция;
- X-lock - монопольная блокировка;
- принцип ACID (англ. «Atomicity, Consistency, Isolation, Durability» - Неделимость (или же Неразрывность, Атомарность), Согласованность, Изолированность, Долговечность);
- протокол ARIES (англ. «Algorithm for Recovery and Isolation Exploiting Semantics» - алгоритм восстановления и изоляции с использованием семантики программирования, то есть начальных смысловых значений операторов, основных конструкций языка и т.п.);
- СУБД DB2;
- СУБД MySQL;
- СУБД Pyrrho;